Exact algorithms for MAX-2SAT and MAX-3SAT via multidimensional matrix multiplication

Yevgheni Petkevich

Master thesis in Informatics

University of Bergen, June 1, 2015

Abstract

In this thesis it is showed how an $O(n^{4-\epsilon})$ algorithm for the CUBE MULTIPLICATION problem (that is defined in the thesis) would imply a faster than naive $O^*(2^{n(1-\frac{\epsilon}{4})})$ algorithm for the MAX-3SAT problem; this algorithm for MAX-3SAT is a generalization of the algorithm for the MAX-2SAT problem which was proposed by Ryan Williams; and CUBE MULTIPLICATION, in turn, is defined as a generalization of the MATRIX MULTIPLICATION problem for three-dimensional arrays. Approaches to find a faster than naive algorithm for CUBE MULTIPLICATION are considered. Though no such algorithm was found using these approaches, it is showed how a variant of the Strassen algorithm for MATRIX MULTIPLICATION could be found using the same approaches. Implementations of these approaches using computer programming and results of computational experiments are discussed.

Keywords: MAX-2SAT, MAX-3SAT, MATRIX MULTIPLICATION, CUBE MULTIPLICATION, ALGORITHMS

Acknowledgements

I found it hard to write acknowledgements. It is so easy to feel the endless gratitude, but it is so hard to express it with words. Nevertheless I will try.

Most of all I want to thank my advisor Daniel Lokshtanov. For awesome support, great inspiration and a lot of valuable comments. It was a great pleasure and priceless experience to work with Daniel.

I also want to thank the Algorithms group. The kind and friendly atmosphere was the first thing I had noticed, and it always surrounded me during my studies.

I wish to thank the department staff as well: I never had any trouble getting answers to any of my questions regarding study formalities.

A very special gratitude goes to the developers of Wikidpad and Zim Wiki, the two wiki-programs I used for my daily life, including the work with this thesis. I cannot imagine now how would I work without having such tool as a personal wiki. I also want to thank Alena and Alesia for helping with English grammar.

I want to thank my friends and teachers for inspiring me to be who I am.

In the end I want to thank the people who were a big part of my life during this period of work, and filled it with happiness, sense and love, and supported me a lot during my work on thesis. Thanks to Katarina, Liina-Liis and Marina.

Contents

Chapter 1

Introduction

1.1 Thesis overview

In this thesis we consider several algorithmic problems. We focus on trying to find a better algorithm for a computational problem, namely CUBE MULTIPLICATION, which is defined in the thesis. We show how an asymptotically faster than naive algorithm for this problem will imply an asymptotically faster than naive algorithm for the known MAX-3SAT problem, for which no faster than brute force algorithms are known to date.

An algorithm has been developed for the MAX-2SAT problem in 2007 by Williams, which is asymptotically faster than a naive algorithm that tries all possible solutions. It is based on reducing MAX-2SAT to the MATRIX MULTIPLICATION problem, whose complexity is a key in obtaining the algorithm with better worst case running time. There exist faster then naive algorithms for MATRIX MULTIPLICATION, and that implies faster than naive algorithms for MAX-2SAT.

As MAX-2SAT and MAX-3SAT are inherently similar, it is possible to build a similar algorithm for MAX-3SAT to the Williams' algorithm for MAX-2SAT. This analogous algorithm is based on reducing MAX-3SAT to CUBE MULTIPLICATION, corresponding to MATRIX MULTIPLICATION in the algorithm for MAX-2SAT, where CUBE MULTIPLICATION is a generalization of MATRIX MULTIPLICATION. But the possibly helpful CUBE MULTIPLICATION problem has not been studied, and therefore no algorithms which are faster then naive are known for it.

If an algorithm which is asymptotically faster than computation by definition for CUBE MULTIPLICATION exists, then, used as a subroutine, it yields an algorithm that is faster than naive for MAX-3SAT. Because MATRIX MULTIPLICATION and CUBE MULTIPLICATION are similar as well,

it may be possible to use the ideas from efficient MATRIX MULTIPLICATION algorithms to make an efficient algorithm for CUBE MULTIPLICATION. The simplest algorithm that improves MATRIX MULTIPLICATION running time is the classic Strassen's algorithm. We will try to generalize the Strassen's algorithm to an algorithm for CUBE MULTIPLICATION. This involves solving a certain large, but finite, puzzle. This thesis accounts how a solution to this puzzle leads to an efficient algorithm for CUBE MULTIPLICATION, and also for MAX-3SAT. The main work of this thesis is to describe our attempts at solving this puzzle. Although we were not able to solve it, we were able to make interesting progress by formulating approaches that easily rediscover the Strassen algorithm.

1.2 Problem description

For known computational problems one usually tries to find efficient algorithms. There are problems which cannot be solved algorithmically; and for solvable problems, it is useful to find better algorithms, both for practical applications and for theoretical implications. Better usually means asymptotically faster. For concrete practical purposes that could mean faster running time, if a new faster algorithm could be implemented. A faster algorithm could be used in other algorithms as a subroutine, thus implying other faster algorithms. The ideas used in the algorithm could be reused elsewhere and improved or expanded, leading towards an even better algorithm for the same problem.

There are polynomial time algorithms and exponential time algorithms, and polynomial time algorithms are usually much faster in practice, especially on large inputs. However, there are problems, called NP-complete, for which we only can hope for exponential algorithms, because it is unlikely that a polynomial algorithm exists for such problems. But there is still a big difference between running times of different exponential time algorithms, and usually one tries to find an exponential algorithm that is as fast as possible.

One of the classical problems in informatics is the SAT problem. Given a boolean formula, it is needed to determine whether the formula could be satisfied with some truth assignment to its variables. This problem is NP-complete. While a naive algorithm for SAT that tries all values of variables has complexity $O^*(2^n)$, there are other better exact exponential time algorithms and heuristic algorithms that could solve the SAT problem in acceptable time. Restricted versions of SAT, the k-SAT problems, in which the boolean formula is in conjunctive normal form and clause size is limited

by k literals are also NP-complete for $k \geq 3$; for $k = 2$ such problem is solvable in polynomial time.

MAX-kSAT is a class of problems, in which the task is to determine whether the given number of clauses could be satisfied by an assignment in a CNF formula with clause size limited by k literals. These problems are NP-complete. A naive algorithm for these problems has $O^*(2^n)$ complexity (checking all possible solutions). Could it be done faster? For MAX-2SAT — yes: with an algorithm by Williams [2007], the MAX-2SAT problem is solvable in $O^*(1.74^n)$ time. For MAX-kSAT where $k \geq 3$ we don't know, and the core question of this thesis is whether we can find a better algorithm, that is, an algorithm with a running time $O^*((2 - \epsilon)^n)$. The trick in the MAX-2SAT algorithm by Williams is to split the original problem into exponentially many subproblems and solve each one with a fast polynomial algorithm. Particularly here, the fast polynomial algorithm is a MATRIX MULTIPLICATION algorithm which is faster than naive.

A naive MATRIX MULTIPLICATION algorithm has complexity $O(n^3)$. Many faster algorithms were discovered for MATRIX MULTIPLICATION, starting with the Strassen's algorithm with running time $O(n^{\log_2 7})$ discovered by Strassen [1969], and the most recent algorithm with running time $O(n^{2.38})$ is discovered by Gall [2014]. The most simple, Strassen's algorithm employs recursive block computation and linear combinations of products of block sums, making possible to do 7 multiplications instead of 8 in each recursive step.

Now consider the MAX-3SAT problem. It is just like MAX-2SAT, except for the number of literals in each clause. Maybe it is possible to create a faster algorithm for it in a similar fashion to the fast algorithm for MAX-2SAT? We could try to do it, and that involves making generalisations in several steps of the fast MAX-2SAT algorithm. One of these generalisations is the cube multiplication, an operation which is defined like matrix multiplication, but on three 3-dimensional arrays. It is a key subroutine in this new MAX-3SAT algorithm, in the sense that improving running time for a CUBE MULTIPLICATION algorithm from naive to something faster gives an improvement in running time for the MAX-3SAT algorithm. If we consider the fact that 2SAT is solvable polynomially and 3SAT is proved to be NP-complete, it could be possible that MAX-3SAT is not improvable, unlike MAX-2SAT.

So then, we arrive at another question, whether we can find a better than a naive $O(n^4)$ algorithm for CUBE MULTIPLICATION, that is an $O(n^{4-\epsilon})$ algorithm. The CUBE MULTIPLICATION problem has not been studied, so initially we don't know a faster algorithm for it and whether a faster algorithm exists at all. CUBE MULTIPLICATION is similar enough to MATRIX MULTIPLICATION to possibly have an algorithm which is based on the same

ideas as Strassen's algorithm, that is recursive block computation and linear combinations of products of block sums. We can define a puzzle to determine whether such algorithm exists. The solution space for this puzzle is very large, much larger than in case of the Strassen's algorithm.

We could try to cope with that puzzle by formulating it as an instance of a problem and trying to develop an efficient algorithm to solve that problem, as well as formulating the puzzle as input for various solvers, such as SAT solvers. SAT solvers take one boolean formula as an input, and give a truth assignment to the variables of the formula if this formula could be satisfied, or say NO otherwise. Various optimizations and heuristics could help us to find an answer to the puzzle, or at least to help us find more information about it.

In this thesis we have tried to formulate necessary problems and proved the correctness of the proposed algorithms. We explain the programming techniques that were used in our attempt to solve the puzzle of finding Strassen-like CUBE MULTIPLICATION algorithm. Although no solution was found during computational experiments, we explain our attempts and their results, our observations and ideas for improvement.

1.3 Note on working on the thesis

The version of the Willams' algorithm for MAX-2SAT presented in the thesis was taken from the book "Exact exponential algorithms" by Fomin and Kratsch [2010]. The formulations for the MAX-3SAT and the SET SPAN problems and the Check-Span-Fast algorithm were informally explained to me by my advisor, and I had formalized them. The SAT formulations for the MATRIX-STRASSEN and CUBE-STRASSEN puzzles, and all the programming work were done by me.

1.4 Thesis structure

In Chapter 2 we present a theoretical foundation that includes important definitions (such as cube and cube multiplication definitions), notation and lemmas, which are used in the main parts.

In Chapter 3 we describe a version of the Williams' algorithm for MAX-2SAT and describe its generalization for MAX-3SAT. We show that an $O(n^{4-\epsilon})$ algorithm for CUBE MULTIPLICATION would imply an algorithm for MAX-3SAT with running time $O^*(2^{n(1-\frac{\epsilon}{4})})$. We also show that this approach works even if there exists an $O(n^{4-\epsilon})$ algorithm for CUBE MULTIPLICATION

over GF(2), because it implies a randomized algorithm for MAX-3SAT with $O^*(2^{n(1-\frac{\epsilon}{4})})$ running time.

In Chapter 4 we describe our approach in finding a faster than naive algorithm for CUBE MULTIPLICATION, yielding the SET SPAN problem and SAT-formulations. We also describe how to discover the Strassen algorithm for MATRIX MULTIPLICATION using the same approach.

In Chapter 5 we describe the programs and the methods that we have developed and used in order to solve two instances of the SET SPAN problem that arise from trying to discover the Strassen's algorithm and the faster CUBE MULTIPLICATION algorithm. We also describe our attempts to use SAT-solvers on the SAT-formulations we have described in Chapter 4.

In Chapter 6 we present our conclusions.

Chapter 2

Theoretical foundation

In this chapter we present relevant theory, terminology and notation. Explanations are simplified and cover only necessary concepts that will be used in successive chapters, with references for further reading. Most of the notation and definitions are common in informatics literature. However, some of the definitions are particular to the thesis, namely *cube* and *cube product* in section 2.6 on page 17 and *k-uniform hypersquare* in section 2.8.2 on page 21.

2.1 Note on indexing

In this thesis we use sometimes a lot of indices, so we will draw attention to the most important properties of indexing that will be used further. If several indices are separated by comma, like $x_{i,j,k}$, they are equivalent to a tuple, as in $x_{(i,j,k)}$. Sometimes several indices are aliased by one (or one index is aliased with several), with encoding rules (explained where used), for example x_h could mean the same as $x_{(i,j,k)}$ when $h = \lambda(i,j,k)$, where $\lambda(i,j,k)$ is a bijection between the domains of (i,j,k) and h. In some cases, the order of indices does not matter, then we index over a set or a multiset (a set that can have several equal elements), like $x_{\{i,j,k\}}$. If two index aliases mean the same, we denote it by putting the symbol $\sim$ between these index aliases: $h \sim (i,j,k)$. Then x_h means the same as $x_{i,j,k}$.

2.2 Algorithms and their complexity

An *algorithm* could be informally defined as a set of instructions for solving any instance of a particular problem, where solving means yielding a correct

answer for a corresponding instance. A common example of an algorithm is the greatest common divisor algorithm. Given two integer numbers, the greatest common divisor algorithm yields a number which has to be the greatest divisor of both given numbers. So we call these two given numbers *input*, and the resulting number *output*, or an *answer*. Usually we consider imperative algorithms, that could be represented as step-by-step operations. For a problem there could be several known algorithms with different properties, and some problems could even be unsolvable from an algorithmic point of view. Among the most important properties of an algorithm are its running time and memory requirements. We describe in this section some of necessary properties of algorithms and different kinds of algorithms, together with a common notation.

Most of the problems we consider in this thesis are *decision problems*. That means that output for such problems can be either YES or NO. For example, consider the parity problem: an integer number is given and the task is to output if the number is odd. Unless stated we assume that a considered problem in this thesis is a decision problem. There are also other, non-decision computational problems. A task of such a problem could be to find the best solution among all possible solutions, to compute a result of a function, or some other task with different possible results than YES or NO. As an example, consider the factorisation problem: an integer number is given and the task is to output all factors of this number.

Usually we consider deterministic algorithms that do the same operations for the same input, and thus yielding the same correct answer.

More in-depth introduction to algorithms could be found in the textbook "Algorithms" by Dasgupta et al. [2006].

2.2.1 Running time and big O notation

Running time, an important algorithm characteristic, is usually measured as a function of a problem's input size. For a problem and an algorithm that solves this problem, for every fixed value of n we look at all instances of the problem of size n and count how many operations the algorithm uses in worst case. Such function can be complicated, hard to compute exactly it and work with. For convenience, we will use notation that will allow us to ignore constant factors and low order terms, that is called big O notation. It gives an understanding of how fast the running time grows when input size is going towards infinity, called asymptotic complexity. For two functions $f, g : N \to N$ we say that $f(n) = O(g(n))$ if there exists some constant $c > 0$ and a number n_0 such that $f(n) < cg(n)$ for all $n > n_0$. We say that an algorithm runs in $O(f(n))$ time if there exists a function $t(n) = O(f(n))$

such that for every input of size n, the algorithm use at most $t(n)$ steps on that input. Asymptotic complexity is one of the main measures of algorithm effectiveness. It shows computation time for a worst case of an algorithm as a function of input size. Asymptotically faster algorithms are usually more practical. Finding a better algorithm for a problem or proving that there is no asymptotically faster algorithm is useful for not only practical reasons, but helps to establish bounds for other algorithms and theoretical problems in informatics.

Usually, when writing running time for an algorithm, we omit lower order terms and constant factors, making notation concise while preserving asymptotic complexity. For example, if an algorithm runs in time $3n^2 + 5n + 2001$ in worst case for an input of size n, we write that it runs in time $O(n^2)$.

A *polynomial* is a function like $poly(n) = c_1 n^{k_1} + \ldots + c_m n^{k_m}$, that is a sum of degrees of n multiplied by constants. We will also use O^*-notation, which is written $f(n) = O^*(g(n))$ and means that $f(n) = O(g(n)poly(n))$. For example, $2^n n^5 = O^*(2^n)$. This notation will be used for comparison of exponential time algorithms.

We will use further in the text the words "faster" and "better" meaning "asymptotically faster". And words "runtime", "time" and "complexity" meaning "asymptotic complexity of a worst case".

More information on Complexity of algorithms could be found in the "Introduction to the Theory of Computation" by Sipser [2012].

2.2.2 Polynomial and exponential time algorithms

Polynomial time algorithms are algorithms that have runtime $O(poly(n))$. Algorithms with running time $O(2^{poly(n)})$ are asymptotically slower and are called *exponential time algorithms*. Although it is usually better to use polynomial time algorithms for any particular problem, because of their practical running time on large inputs, sometimes there are no known polynomial algorithms for such problem. Then it is natural to find as fast exponential time algorithm as possible. However, for particular polynomial and exponential time algorithms, on some input sizes (small enough) the exponential algorithm could be faster to run in practice. For hard problems without a known polynomial time algorithm it could be useful to find efficient exponential algorithms to make bigger instance solvable in practice. Another reason to find better exponential time algorithms for a problem is to improve known time bounds for theoretical implications. Such research can help to understand the problem better and used algorithmic techniques could be improved further and used in other algorithms for other problems as well.

The problems that we will consider in this thesis do usually have a finite number of possible solutions. A naive algorithm for such problems usually checks every possible solution from the space of possible solutions and thus its running time is proportional to the size of the possible solution space. Therefore it is a question for such problems whether it is necessary to go through all the possible solutions or we can avoid it by somehow discarding sets of wrong possible solutions.

One can find more information regarding exponential time algorithms in the book by Fomin and Kratsch [2010].

Problems are also classified by their computational complexity. The classes we are interested in are P and NP. P is a class of problems that have a polynomial time algorithm, and NP is a class of problems for which solution can be verified using a polynomial time algorithm. $P \subseteq NP$, but it is not known if $P = NP$ or $P \subset NP$.

Some problems in NP class are called *NP-complete*. That means that they are the hardest in this class. Any other problem in NP can be reduced to an NP-complete problem. Problems that are NP-Complete are considered hard to solve, because it is unlikely that a polynomial time algorithm for solving such a problem exists. However, exponential time algorithms still can be used to solve such problem. And among these exponential time algorithms some can be substantially faster than others.

2.2.3 Randomized algorithms

If we allow an algorithm to make random decisions during its run we call such algorithm a *randomized algorithm*. Consider such algorithm as having access to an an oracle that can give a uniformly random number for any requested range. A randomized algorithm, unlike a deterministic algorithm, may behave differently on the same input. It could always yield a correct answer, or it could yield a correct answer with some probability. A running time of a randomized algorithm may also vary for the same input. Such algorithms can be useful for many reasons, and one can find a better introduction in the "Algorithm design" book by Kleinberg and Tardos [2006].

In this thesis we consider randomized algorithms whose running time will always be upper bounded by a function of an input size regardless of random choices made by such algorithm. We also consider only randomized algorithms for decision problems. However these algorithms can yield an incorrect answer with some probability. We only consider in this thesis such randomized algorithms that always yield NO answer for a NO instance and will yield a correct YES answer with a probability (or incorrect NO with a probability) in case of YES instance. That is the only possible case for a

randomized algorithm to yield an incorrect answer is to yield a NO answer for a YES instance.

Consider a randomized algorithm **A** that yields a correct answer with a probability at least p if the answer is YES and which is always correct when the answer is NO. If we run this algorithm **A** k times, then with probability $(1-p)^k$ at most it will yield NO after all k runs if the answer is YES, otherwise outputting a correct answer at least once. Then, we can construct another algorithm as follows:

Algorithm B. Run **A** k times. If at least once the output of **A** is YES, output YES. Otherwise, output NO.

Then the algorithm **B** yields a correct answer with a probability at least $(1-(1-p)^k)$ and its running time is k times greater than the running time of **A**. For any probability p, $0 < p < 1$ there exists an integer number k such that $(1-(1-p)^k)$ is arbitrarily close to one. That means, if there exists a randomized algorithm that yields a correct YES answer with a probability p for a problem, $0 < p < 1$, then there exists an algorithm that solves this problem with arbitrarily high probability at the cost of increased running time.

Consider a small example of a randomized algorithm. We are given an array of n integer numbers and the task is to find out if at least one of the numbers is odd. A simple randomized algorithm would be:

Algorithm C. Pick a number randomly from the array. If it is odd, output YES, otherwise output NO.

The algorithm **C** always yields a right answer if an instance has no odd numbers, and a right answer with probability at least $1/n$ if there is at least one odd number in the array. Thus, with at least $1/n$ probability we get a correct answer. By running the algorithm **C** n times we can achieve a probability $(1-(1-1/n)^n) > (1-1/\epsilon) > 0.63$ for any n. If we run it $10n$ times we can achieve a probability $(1-(1-1/n)^{10n}) > (1-(1/\epsilon)^{10}) > 0.99995$ for any n.

If an algorithm **D** yields a right answer with a probability $p = 1/(2^n)$, to achieve a probability $(1-1/\epsilon) > 0.63$ by running it several times, we need to run it 2^n times.

So there is a tradeoff between the resulting probability and the running time. We will work with algorithms whose probability of a correct answer will always be a constant.

2.3 Boolean formulas and satisfiability problems

2.3.1 Boolean formulas

Boolean formulas are formulas that contain *boolean variables*, that is variables that could have one of two values: TRUE and FALSE (or "1" and "0", or YES and NO). A boolean variable is a formula itself and its value is the same as value of the variable. Larger boolean formulas are constructed using *logical operators* in a similar fashion to arithmetic operators.

Consider a boolean formula ϕ. The operator $\neg$ is called *negation*, and a boolean formula $\neg\phi$ has value TRUE if the value of ϕ is FALSE and vice versa. Consider boolean formulas ϕ and ψ. The binary operator $\vee$ is called *disjunction*, and a boolean formula $\phi \vee \psi$ has value FALSE if values of both ϕ and ψ are FALSE, and TRUE otherwise. The binary operator $\wedge$ is called *conjunction*, and a boolean formula $\phi \wedge \psi$ has value TRUE if values of both ϕ and ψ are TRUE, and FALSE otherwise. The binary operator $\oplus$ is called *exclusive disjunction*, and a boolean formula $\phi \oplus \psi$ has value TRUE if values of both ϕ and ψ are different, and FALSE otherwise. Operators are applied in the same way as in arithmetic, where $\neg$ has the biggest priority, $\wedge$ has a medium priority, and $\vee$ and $\oplus$ have the lowest priority. Parentheses can also be used to alter the priority of operators, in the same way as in arithmetic. An example of a boolean formula is:

$$\phi_1 = (\neg x_1 \vee (x_2 \vee x_3) \wedge x_4) \oplus x_5. \tag{2.1}$$

A variable x or its negation $\neg x$ is called a *literal*. A disjunction of several literals $(x_1 \vee \ldots \vee x_k)$ is called a *clause*. A clause has value TRUE if at least one of x_i literals has TRUE value, and FALSE otherwise. A boolean formula that is a conjunction of several clauses is called a *boolean formula in conjunctive normal form* or *CNF formula*. An example is:

$$\phi_2 = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_2) \wedge (x_3 \vee x_4). \tag{2.2}$$

A *truth assignment* α to variables is an assignment of 1 (TRUE) or 0 (FALSE) value to these variables. The number of possible truth assignments to n variables is 2^n. For the variables in ϕ_2 in the example above, one of the 16 possible truth assignments could be $\alpha = (x_1 := 1, x_2 := 0, x_3 := 0, x_4 := 1)$. This truth assignment evaluates the formula ϕ_2 to TRUE.

A formula ϕ is called *satisfiable* if there exists a truth assignment that evaluates this formula to TRUE. Otherwise the formula is called *unsatisfiable*. If 1 (TRUE) or 0 (FALSE) value is assigned only to some of the variables

in a formula we call this a *partial truth assignment*. A clause is *satisfied* by a truth assignment or a partial truth assignment if all variables in that clause has value assigned to them and the assignment evaluates this clause to TRUE. So the formula ϕ_2 from the example above is satisfiable.

A special case of a CNF formula is a *k-CNF formula*, a formula in which each clause contain at most k literals. Formula $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_2) \wedge (x_3 \vee x_4)$ is a 3-CNF formula, but not a 2-CNF formula.

More about boolean formulas could be read in a book by Whitesitt [1995].

2.3.2 Satisfiability problems

A classic algorithmic problem is the Satisfiability problem (SAT).

Problem 1 (SAT)**.**
Input: A boolean formula ϕ having n variables and m clauses.
Task: Determine whether there exists a truth assignment to the variables of ϕ that satisfies all the clauses in ϕ.

The SAT problem has proven to be NP-complete by Cook [1971] and by Levin [1973].

If we add a restriction for the number of literals in clauses, we get the following class of problems.

Problem 2 (k-SAT)**.**
Input: A k-CNF formula ϕ having n variables and m clauses.
Task: Determine whether there exists a truth assignment to the variables of ϕ that satisfies all clauses in ϕ.

With $k = 2$, namely 2SAT, there exists a polynomial time algorithm for it, for example an algorithm proposed by Krom [1967]. With $k = 3$, namely 3SAT, the problem was proven to be NP-complete by Karp [1972]. Since for $k > 3$ k-SAT problems are not easier than 3SAT and not harder than SAT, they are also NP-complete.

A generalization of SAT is the maximum satisfiability problem (MAX-SAT).

Problem 3 (MAX-SAT)**.**
Input: A CNF formula ϕ having n variables and m clauses, and integer number $\tilde{k}$.
Task: Determine whether there exists a truth assignment to the variables of ϕ that satisfies at least $\tilde{k}$ clauses.

Since a solution to MAX-SAT leads to a solution to SAT, it is also an NP-complete problem.

With restriction for the number of literals in clauses, we get the following class of problems.

Problem 4 (MAX-kSAT)**.**
Input: A k-CNF formula ϕ having n variables and m clauses, and integer number $\tilde{k}$.
Task: Determine whether there exists a truth assignment to the variables of ϕ that satisfies at least $\tilde{k}$ clauses.

Garey et al. [1976] have proved that MAX-2SAT is NP-complete. Since MAX-2SAT can be reduced to MAX-kSAT, MAX-kSAT is NP-complete for all $k > 2$.

For SAT and MAX-SAT no algorithms better than $O(2^n)$ are known to date. Paturi et al. [2005] have discovered exponential time algorithms for k-SAT with better than naive running time for all $k \geq 3$, particularly $O(2^{0.446n})$ for 3SAT.

For a long time it was an open problem if there exists a better than $O(2^n)$ algorithm, until Williams [2007] has found a faster algorithm with running time $O(2^{(\omega/3)n})$, where ω is the exponent for matrix multiplication. For MAX-kSAT with $k \geq 3$ it is still an open problem if there exists a more efficient algorithm than than a naive $O(2^n)$ algorithm.

2.4 Rings and the GF(2) field

An algebraic *ring* is a set of elements with special operations defined for its elements — *addition* and *multiplication* —, and two special elements, *zero* and *one*. Addition and multiplication in a ring are generalisations of the corresponding arithmetic operations, and zero and one elements are generalisations of the corresponding elements in arithmetic.

GF(2) is a ring with two elements, 0 and 1. Addition and multiplication in GF(2) are defined as arithmetical addition and multiplication modulo 2. That means $1 + 1 = 0$ in GF(2).

GF(2) is in fact a *field* (another algebraic object), but we will not use this fact in this thesis.

One can find more about rings and fields in a book by Jacobson [2012].

2.5 Matrices and matrix multiplication

An $m \times n$ *matrix over a ring* E is a two-dimensional array with m rows and n columns, such that each element of the array is an element of E. A matrix with the same number of rows and columns is called *square matrix*. We can call an $m \times 1$ matrix an *m-dimensional vector*. Here are examples of a 3×4 matrix over real numbers and a square 3×3 matrix over GF(2):

$$\begin{bmatrix} 5.25 & 5 & 0.44 & 12 \\ 0 & 1 & 5 & 555 \\ 33.3 & 1 & 654 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix} \tag{2.3}$$

We denote the element in the i-th row and j-th column of a matrix A by $A_{i,j}$. For example, a 3×3 matrix A:

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,2} \\ A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,1} & A_{3,2} & A_{3,3} \end{bmatrix} \tag{2.4}$$

Several operations are defined for matrices. The sum $A + B$ of two $m \times n$ matrices A and B is calculated element-wise:

$$(A + B)_{i,j} = A_{i,j} + B_{i,j}, \quad 1 \leq i \leq m, 1 \leq j \leq n. \tag{2.5}$$

The subtraction of two matrices is defined in the same way. The multiplication cA of an $m \times n$ matrix A by a number c is also calculated element-wise:

$$(cA)_{i,j} = c \cdot A_{i,j}, \quad 1 \leq i \leq m, 1 \leq j \leq n. \tag{2.6}$$

Another operation defined for matrices is *matrix multiplication*. The matrix multiplication of an $m \times n$ matrix A and an $n \times k$ matrix B is defined as an $m \times k$ matrix (AB) with elements calculated as follows:

$$(AB)_{i,j} = \sum_{l=1}^{n} A_{i,l} \cdot B_{l,j}, \quad 1 \leq i \leq m, 1 \leq j \leq k \tag{2.7}$$

Note that matrix multiplication is only defined for two matrices when the number of columns of the first matrix is the same as the number of rows of the second matrix. And in general $AB \neq BA$, so the order of matrices matters. Matrix multiplication has the following properties (called *distributive properties*):

$$((AB)C) = (AC) + (BC) \tag{2.8}$$

$$(A(BC)) = (AB) + (AC) \tag{2.9}$$

We denote by a *block matrix* a matrix which is interpreted as having been partitioned into sections called *blocks*. Each block is a matrix itself and is denoted by $A^{i,j}$. For $n = mk$, an $n \times n$ matrix A over E can be thought of as an $m \times m$ matrix where each element is an $k \times k$ matrix over E. For example:

$$A = \begin{bmatrix} A^{1,1} & \cdots & A^{1,m} \\ \vdots & \ddots & \vdots \\ A^{m,1} & \cdots & A^{m,m} \end{bmatrix}, \tag{2.10}$$

where each block $A^{p,q}$ is a $k \times k$ matrix:

$$A^{p,q} = \begin{bmatrix} A^{p,q}_{1,1} & \cdots & A^{p,q}_{1,k} \\ \vdots & \ddots & \vdots \\ A^{p,q}_{k,1} & \cdots & A^{p,q}_{k,k} \end{bmatrix}. \tag{2.11}$$

We use both superscript and subscript indices to denote an element of a block matrix, where superscript is used to index blocks, and subscript is used to index elements inside blocks, like $A^{p,q}_{s,t}$. Thus, $A_{i,j}$ and $A^{p,q}_{s,t}$ denote the same element of the matrix A from the example above, if

$$\begin{array}{ll} i = (p-1)k + s, & 1 \le p \le m, 1 \le s \le k, \\ j = (q-1)k + t, & 1 \le q \le m, 1 \le t \le k \end{array}. \tag{2.12}$$

We denote by MATRIX MULTIPLICATION the computational problem where we are given two $n \times n$ matrices A and B as input, and the task is to compute (AB).

Directly applying the definition of matrix multiplication yields an algorithm for MATRIX MULTIPLICATION with an $O(n^3)$ running time. However, there exist asymptotically faster than $O(n^3)$ algorithms for it, in particular, the exponent here can be less than 3. Any algorithm for matrix multiplications should calculate n^2 elements of the resulting matrix, so that exponent cannot be smaller than 2. We denote by ω such exponent. It is an open problem, what is the fastest algorithm for MATRIX MULTIPLICATION. The first of fast matrix multiplication algorithms was discovered by Strassen [1969]. The matrix multiplication is calculated recursively, by multiplying blocks of 2×2 block matrices as elements of 2×2 matrix at each recursive step. The trick is to use 7 multiplications in each step instead of 8. Let's see this algorithm in more details.

So we want to find matrix multiplication (AB) of $n \times n$ matrices A and B. Matrix multiplication can be computed using block partitioning. Without loss of generality we assume that $n = mk$ (otherwise we enlarge matrices,

filling new values with zeros; the instance size will be increased by constant factor only). We partition A and B into $m \times m$ blocks. It is easy to prove (and we will prove the same trait later for *cube multiplication* in lemma 8 on page 40), that if we compute matrix multiplication of A and B treating each block as element and element multiplication as matrix multiplication, we'll get a block matrix AB which is equal to a matrix multiplication of A and B:

$$\begin{bmatrix} (AB)^{1,1} & \cdots & (AB)^{1,m} \\ \vdots & \ddots & \vdots \\ (AB)^{m,1} & \cdots & (AB)^{m,m} \end{bmatrix} = \begin{bmatrix} A^{1,1} & \cdots & A^{1,m} \\ \vdots & \ddots & \vdots \\ A^{m,1} & \cdots & A^{m,m} \end{bmatrix} \begin{bmatrix} B^{1,1} & \cdots & B^{1,m} \\ \vdots & \ddots & \vdots \\ B^{m,1} & \cdots & B^{m,m} \end{bmatrix} = \\ = \begin{bmatrix} \sum_{l=1}^{n} A^{1,l} B^{l,1} & \cdots & \sum_{l=1}^{n} A^{1,l} B^{l,m} \\ \vdots & \ddots & \vdots \\ \sum_{l=1}^{n} A^{m,l} B^{l,1} & \cdots & \sum_{l=1}^{n} A^{m,l} B^{l,m} \end{bmatrix} \quad (2.13)$$

This gives us a recursive algorithm for MATRIX MULTIPLICATION where in order to compute the multiplication of two $n \times n$ matrices we compute m^3 matrix multiplications of two $\frac{n}{m} \times \frac{n}{m}$ matrices. Its running time is upper bounded by the following recurrence relation:

$$T(1) = O(1), \quad (2.14)$$
$$T(n) = m^3 T(n/m) + mO(n^2), \quad (2.15)$$

where m^3 recursive calls to MATRIX MULTIPLICATION of blocks are accounted in the first term of the right side in formula (2.15), and summation of them element-wise in the second term. Considering m as a constant, an application of the master theorem (refer to the book by Dasgupta et al. [2006] for the explanation and proof of the master theorem) shows that this recurrence gives us an $O(n^{\log_m^{(m^3)}}) = O(n^3)$ algorithm, which is the same complexity as of a naive algorithm. If we let $m = 2$, we need to make 8 recursive matrix multiplications in that algorithm each time. Strassen has found how to make 7 multiplications instead of 8, thus changing $\log_2 8$ exponent to $\log_2 7$, yielding an algorithm with complexity $O(n^{\log_2 7})$. He have used the following

formulas for the 7 multiplications:

$$M_1 = (A^{1,1} + A^{2,2})(B^{1,1} + B^{2,2}), \tag{2.16}$$
$$M_2 = (A^{2,1} + A^{2,2})B^{1,1}, \tag{2.17}$$
$$M_3 = A^{1,1}(B^{1,2} - B^{2,2}), \tag{2.18}$$
$$M_4 = A^{2,2}(-B^{1,1} + B^{2,1}), \tag{2.19}$$
$$M_5 = (A^{1,1} + A^{1,2})B^{2,2}, \tag{2.20}$$
$$M_6 = (-A^{1,1} + A^{2,1})(B^{1,1} + B^{1,2}), \tag{2.21}$$
$$M_7 = (A^{1,2} - A^{2,2})(B^{2,1} + B^{2,2}), \tag{2.22}$$

and the following formulas to compute the blocks of AB:

$$(AB)^{1,1} = M_1 + M_4 - M_5 + M_7, \tag{2.23}$$
$$(AB)^{2,1} = M_2 + M_4, \tag{2.24}$$
$$(AB)^{1,2} = M_3 + M_5, \tag{2.25}$$
$$(AB)^{2,2} = M_1 + M_3 - M_2 + M_6. \tag{2.26}$$

Since then, many improvements have been made, and to date, the asymptotically fastest algorithm for matrix multiplication has complexity $O(n^{2.38})$, discovered by Gall [2014].

2.6 Cubes and cube multiplication

We denote by an $n \times n \times n$ *cube* or *n-cube* over a ring E a 3-dimensional array with n rows, columns and planes, such that each element of the array is an element of E. One can see an n-cube as a generalisation of an $n \times n$ square matrix. Given such an n-cube A, the element of A in row i, column j and plane k is denoted by $A_{i,j,k}$:

$$A = \left[\begin{bmatrix} A_{1,1,1} & \cdots & A_{1,n,1} \\ \vdots & \ddots & \vdots \\ A_{n,1,1} & \cdots & A_{n,n,1} \end{bmatrix}, \cdots, \begin{bmatrix} A_{1,1,n} & \cdots & A_{1,n,n} \\ \vdots & \ddots & \vdots \\ A_{n,1,n} & \cdots & A_{n,n,n} \end{bmatrix} \right] \tag{2.27}$$

Several operations are defined for cubes. The *sum* $A + B$ of two n-cubes A and B is calculated element-wise:

$$(A + B)_{i,j,k} = A_{i,j,k} + B_{i,j,k}, \quad 1 \leq i \leq m, 1 \leq j \leq n, 1 \leq k \leq n. \tag{2.28}$$

The *subtraction* of two cubes is defined in the same way. The *multiplication* cA of a n-cube A by a number c is also calculated element-wise:

$$(cA)_{i,j,k} = c \cdot A_{i,j,k}, \quad 1 \leq i \leq m, 1 \leq j \leq n, 1 \leq k \leq n. \tag{2.29}$$

Definition 1 (Cube multiplication)**.** Let A, B and C be n-cubes. By the *cube multiplication* of n-cubes A, B and C (in that order) we define an n-cube (ABC) such that:

$$(ABC)_{i,j,k} = \sum_{l=1}^{n} A_{l,j,k} B_{i,l,k} C_{i,j,l}, \quad i,j,k = 1 \dots n. \tag{2.30}$$

The cube multiplication is a generalization of the matrix multiplication for square matrices. Note that the cube multiplication is defined for three cubes, so it is a ternary operation, and that the order of operands matters.

We will denote by CUBE MULTIPLICATION the computational problem where we are given three n-cubes A, B, C as input, and the task is to compute (ABC). Directly applying the definition of cube multiplication yields an algorithm for CUBE MULTIPLICATION with running time $O(n^4)$.

To our knowledge, the CUBE MULTIPLICATION problem has not been studied.

Now we prove a distributive property for cube multiplication.

Lemma 1 (Distributivity over cube addition)**.** *Let A, B, C and D be n-cubes. Then the following statements are correct:*

$$((A+B)CD) = (ACD) + (BCD) \tag{2.31}$$
$$(A(B+C)D) = (ABD) + (ACD) \tag{2.32}$$
$$(AB(C+D)) = (ABC) + (ABD) \tag{2.33}$$

Proof. Let's prove the first equality:

$$\begin{aligned} ((A+B)CD)_{i,j,k} &= \sum_{l=1}^{n} (A_{l,j,k} + B_{l,j,k}) C_{i,l,k} D_{i,j,l} = \\ &= \sum_{l=1}^{n} (A_{l,j,k} C_{i,l,k} D_{i,j,l} + B_{l,j,k}) C_{i,l,k} D_{i,j,l}) = \\ &= (ACD)_{i,j,k} + (BCD)_{i,j,k}, \\ i,j,k = &\quad 1, \dots, n \end{aligned} \tag{2.34}$$

The second and third equality can be proved in the same manner. □

Corollary 1. *If $A_1, \dots, A_k$, $B_1, \dots, B_l$, $C_1, \dots, C_m$ are cubes, then*

$$((A_1 + \dots + A_k)(B_1 + \dots + B_l)(C_1 + \dots + C_m)) = \sum_{p=1}^{k} \sum_{q=1}^{l} \sum_{r=1}^{m} (A_p B_q C_r) \tag{2.35}$$

We denote by a *block cube* a cube which is interpreted as having been partitioned into sections called *blocks*. Each block is a cube itself and is denoted by $A^{i,j,k}$. For $n = mo$, an n-cube A over E can be thought of as an m-cube where each element is an o-cube over E. For example:

$$A = \left[\begin{bmatrix} A^{1,1,1} & \cdots & A^{1,m,1} \\ \vdots & \ddots & \vdots \\ A^{m,1,1} & \cdots & A^{m,m,1} \end{bmatrix}, \cdots, \begin{bmatrix} A^{1,1,m} & \cdots & A^{1,m,m} \\ \vdots & \ddots & \vdots \\ A^{m,1,m} & \cdots & A^{m,m,m} \end{bmatrix} \right], \tag{2.36}$$

and each block $A^{p,q,r}$ is an o-cube:

$$A^{p,q,r} = \left[\begin{bmatrix} A^{p,q,r}_{1,1,1} & \cdots & A^{p,q,r}_{1,o,1} \\ \vdots & \ddots & \vdots \\ A^{p,q,r}_{o,1,1} & \cdots & A^{p,q,r}_{o,o,1} \end{bmatrix}, \cdots, \begin{bmatrix} A^{p,q,r}_{1,1,o} & \cdots & A^{p,q,r}_{1,o,o} \\ \vdots & \ddots & \vdots \\ A^{p,q,r}_{o,1,o} & \cdots & A^{p,q,r}_{o,o,o} \end{bmatrix} \right]. \tag{2.37}$$

We use both superscript and subscript indices to denote an element of a block cube, where superscript is used to index blocks, and subscript is used to index elements inside blocks, like $A^{p,q,r}_{s,t,u}$. Thus, $A_{i,j,k}$ and $A^{p,q,r}_{s,t,u}$ denote the same element of the cube A from the example above, if

$$\begin{aligned} i &= (p-1)o + s, \quad 1 \le p \le m, \quad 1 \le s \le o, \\ j &= (q-1)o + t, \quad 1 \le q \le m, \quad 1 \le t \le o, \\ k &= (r-1)o + u, \quad 1 \le r \le m, \quad 1 \le u \le o. \end{aligned} \tag{2.38}$$

2.7 Basic Linear Algebra

A *vector space* is a set of objects called *vectors*, which may be added together and multiplied by numbers, called *scalars* in this context.

We will be working with the vector space $\mathbf{F}_2^n$, namely the set of tuples of length n where each element of tuple is an element of GF(2). We will call such vectors *n-dimensional GF(2) vectors*. By v_i we define the i-th element of a vector v.

The vector $[0, \ldots, 0]$ is called *zero vector*.

Let u and v be elements of $\mathbf{F}_2^n$:

$$u = [u_1, \ldots, u_n], \tag{2.39}$$

$$v = [v_1, \ldots, v_n]. \tag{2.40}$$

Addition of elements of $\mathbf{F}_2^n$ and *multiplication by scalar* from GF(2) are element-wise:

$$v + u = [v_1 + u_1, \ldots, v_n + u_n], \tag{2.41}$$

$$\alpha v = [\alpha v_1, \ldots, \alpha v_n] \tag{2.42}$$

A sum of several vectors multiplied by scalars each, $\alpha_1 v_1 + \ldots + \alpha_k v_k$, is called a *linear combination*.

The set of vectors $T = \{v_1, \ldots, v_k\}$ is called *linearly independent* if the equation

$$\alpha_1 v_1 + \ldots + \alpha_k v_k = \mathbf{0}, \tag{2.43}$$

where $\mathbf{0}$ is the zero vector, can only be satisfied by $\alpha_i = 0$ for $i = 0, \ldots, k$. Otherwise the set T is called *linearly dependent*.

A *subspace* of a vector space V is a subset U of V, such that any vector that can be expressed as a linear combination of elements from U is also an element of U.

For a set T of vectors, let $span(T)$ be the set of all vectors that can be expressed as a linear combination of vectors from T. We call it the *span of* T. A $span(T)$ is a subspace of $\mathbf{F}_2^n$.

A set B of linearly independent vectors is called a *basis* of a vector space $\mathbf{F}$ if every vector in $\mathbf{F}$ can be expressed as a linear combination of vectors from B. A vector space can have different bases. A number of vectors in a basis of $\mathbf{F}$ is called rank of $\mathbf{F}$. Every vector in $\mathbf{F}$ can be uniquely expressed as a linear combination of basis vectors.

For a vector space $\mathbf{F}$ of rank k any set of k linearly independent vectors is a basis of $\mathbf{F}$.

The following lemma will be used later in our algorithm for the SET SPAN problem. It is a basic fact from linear algebra, we include a proof for completeness. For more information regarding linear algebra one can see a book by Lay [2012].

Lemma 2. *If a set $M = \{m_1, \ldots, m_k\}$ is a set of k linearly independent vectors, and all vectors in a set of linear independent vectors $R = \{r_1, \ldots, r_l\}$ are in $span(M)$, then there exists a subset $P \subseteq M$, such that $P \cup R$ is a basis of $span(M)$.*

Proof. We prove it by induction. A base case is when $R = M$. Then $P = \varnothing$ and R is the basis.

Assume that the statement of the lemma holds for any R of size $l + 1$. Let's prove that it holds for $R = \{r_1, \ldots, r_l\}$ of size l.

Because $span(R) \subset span(M)$ there exists a vector m in M that is not in the span of R (otherwise M would be a subset of $span(R)$). If we add this vector m to R, we get a set $R' = R \cup m$ of $l+1$ linearly independent vectors. By our assumption for R' there exists a set $P' \subseteq M$ such that $P' \cup R'$ is a basis of $span(M)$. Then, if we let $P = P' \cup m$, then $P \cup R = P' \cup m \cup R = P' \cup R'$ is a basis of $span(M)$. □

2.8 Graphs and hypergraphs

2.8.1 Graphs

A *graph* is an ordered pair $G = (V, E)$, where V is a set of *nodes* and E is a set of *edges* which are 2-element subsets of V.

For example, $G_1 = (\{a, b, c, d, e\}, \{\{a, b\}, \{a, c\}, \{b, d\}, \{b, c\}, \{a, d\}\})$ is a graph with five nodes and five edges.

We call a graph $I = (V_I, E_I)$ a *subgraph* of a graph $G = (V, E)$ if $V_I \subseteq V$ and $E_I \subseteq E$. We call a subgraph T of a graph G a *triangle* if T is of the form $(\{a, b, c\}, \{\{a, b\}, \{b, c\}, \{c, a\}\})$, that is if it is a 3-node subgraph that has all possible edges. We also denote such triangle by *abc* for short. In the example graph G_1, *abc* is a triangle.

If every edge $e \in E$ in a graph $G = (V, E)$ has a number assigned to it, that we call *weight*, we call the graph G a *weighted graph*. Each edge could have several different independent weights. We name such weights with a preceding Greek letter, like this: π-weight, ρ-weight, etc. And we denote weights of an edge $\{a, b\}$ like this: $\pi(\{a, b\})$, $\rho(\{a, b\})$, etc.

If there is a triangle *abc* in a weighted graph G, then by *weight of the triangle abc* we denote the sum of weights of all its edges. For example, if edges in G have π-weight, then the triangle *abc* also has π-weight:

$$\pi(abc) = \pi(\{a, b\}) + \pi(\{b, c\}) + \pi(\{c, a\}). \tag{2.44}$$

2.8.2 Hypergraphs

A *hypergraph* is an ordered pair $H = (V, E)$ where V is a set of *nodes* and E is a set of *hyperedges*, which are non-empty subsets of V. A *k-uniform hypergraph* is a hypergraph such that all its hyperedges have size k. So a 2-uniform hypergraph is a graph, and hypergraphs can be seen as a generalization of graphs.

For example, $H_1 = (\{a, b, c, d, e\}, \{\{a, b\}, \{a, b, c\}, \{b, d\}, \{e\}\})$ is a hypergraph with five nodes and four hyperedges. And $H_2 = (\{a, b, c, d, e, f\}, \{\{a, b, c\}, \{b, c, d\}, \{c, d, a\}, \{d, a, b\}, \{c, e, f\}, \{e, f, b\}\})$ is a 3-uniform hypergraph with six nodes and six hyperedges.

We call a hypergraph $I = (V_I, E_I)$ a *subhypergraph* of a hypergraph $H = (V, E)$ if $V_I \subseteq V$ and $E_I \subseteq E$. We call a subhypergraph S of a 3-uniform hypergraph H a *hypersquare* if S is of the form $(\{a, b, c, d\}, \{\{a, b, c\}, \{b, c, d\}, \{c, d, a\}, \{d, a, b\}\})$, that is if it is a 4-node 3-uniform subhypergraph that has all possible hyperedges. We also denote such hypersquare by *abcd* for short. In the example 3-uniform hypergraph H_2, *abcd* is a hypersquare.

If every hyperedge $e \in E$ in a hypergraph $H = (V, E)$ has a number assigned to it, that we call *weight*, we call the hypergraph H a *weighted hypergraph*. Each hyperedge could have several different independent weights. We name such weights with a preceding Greek letter, like this: π-weight, ρ-weight, etc. And we denote weights of a hyperedge $\{a, b, c\}$ like this: $\pi(\{a, b, c\})$, $\rho(\{a, b, c\})$, etc.

If there is a hypersquare *abcd* in a weighted hypergraph H, then by *weight of the hyperquare abcd* we denote the sum of weights of all its hyperedges. For example, if hyperedges in H have π-weight, then the hypersquare *abcd* also has π-weight:

$$\pi(abcd) = \pi(\{a, b, c\}) + \pi(\{b, c, d\}) + \pi(\{c, d, a\}) + \pi(\{d, a, b\}). \quad (2.45)$$

2.9 The Isolation lemma

The isolation lemma, proposed and proved by Mulmuley et al. [1987] will be useful for us to reduce a problem for which several possible solutions could exist to another problem that has at most one solution with high probability. When we speak about several possible solutions for a decision problem, we mean the following. Consider a MAX-kSAT problem. An instance of that problem could have several different truth assignments that satisfy at least $\tilde{k}$ clauses. Then, the reduction succeeds with high probability p in the following sense. If there is a YES-instance of MAX-kSAT then there will be a YES-instance of a reduced problem, and if there is a NO-instance of MAX-kSAT then there will be a NO-instance of a reduced problem. Among all truth assignments that satisfy at least k clauses, with the probability p there will be a unique truth assignment with the minimum weight.

Lemma 3. *Let n and N be positive integers, and let $\mathcal{F}$ be an arbitrary family of subsets of the set $S = \{1, \ldots, n\}$. Suppose each element $x \in \{1, \ldots, n\}$ in S receives an integer weight $\rho(x)$, each of which is chosen independently and uniformly at random from $\{1, \ldots, N\}$. The weight of a set S in $\mathcal{F}$ is defined as*

$$\rho(S) = \sum_{x \in S} \rho(x).$$

Then, with probability at least $1 - \frac{n}{N}$, there is a unique set in $\mathcal{F}$ that has the minimum ρ-weight among all sets of $\mathcal{F}$.

Proof. Suppose we have fixed the weights of all elements except an element x. Then x has a *threshold* weight α, such that if the weight $\rho(x)$ of x is greater than α, then it is not contained in any minimum-weight subset, and

if $\rho(x) \leq \alpha$, then it is contained in some sets of minimum weight. Further, observe that if $\rho(x) < \alpha$, then every minimum-weight subset must contain x (since, when we decrease $\rho(x)$ from α, sets that do not contain x do not decrease in weight, while those that contain x do). Thus, ambiguity about whether a minimum-weight subset contains x or not can happen only when its weight is exactly equal to its threshold; in this case we will call x *singular*. Now, as the threshold of x was defined only in terms of the weights of the other elements, it is independent of $\rho(x)$, and therefore, as $\rho(x)$ is chosen uniformly from $\{1, \ldots, N\}$,

$$P[x \text{ is singular}] = P[\rho(x) = \alpha] \leq 1/N \tag{2.46}$$

and the probability that some x is singular is at most n/N. As there is a unique minimum-weight subset if no element is singular, the lemma follows. □

Chapter 3

MAX-kSAT algorithms

We know that MAX-kSAT problems are NP-complete. For all of them there exists a naive algorithm with running time $O^*(2^n)$, that checks all possible truth assignments. Currently, we don't know any faster algorithms for these problems except for the faster MAX-2SAT algorithm discovered by Williams [2007].

The algorithm of Williams reduces MAX-2SAT to MATRIX MULTIPLICATION in such a way that an $O(n^{3-\epsilon})$ algorithm for MATRIX MULTIPLICATION implies an $O^*(2^{n(1-\frac{\epsilon}{3})})$ algorithm for MAX-2SAT. Thus, a naive $O(n^3)$ algorithm for MATRIX MULTIPLICATION gives an alternative $O^*(2^n)$ algorithm for MAX-2SAT. Anything faster for MATRIX MULTIPLICATION gives a better than $O^*(2^n)$ algorithm for MAX-2SAT. Applying the best known algorithm for MATRIX MULTIPLICATION by Gall [2014] ($O(n^{2.38})$) gives an $O^*(2^{0.80n})$ algorithm for MAX-2SAT. Applying the Strassen's $O(n^{2.81})$ algorithm for MATRIX MULTIPLICATION gives an $O^*(2^{0.94n})$ algorithm for MAX-2SAT.

We aim to make a better MAX-3SAT algorithm in a similar manner. We will give a reduction from MAX-3SAT to CUBE MULTIPLICATION based on the same ideas that were used in the MAX-2SAT algorithm by Williams, such that a naive algorithm for CUBE MULTIPLICATION gives us an alternative $O^*(2^n)$ algorithm for MAX-3SAT, and any better algorithm for CUBE MULTIPLICATION gives us a better algorithm for MAX-3SAT. The algorithm does not, at this point, beat $O^*(2^n)$ for MAX-3SAT but it shows that an algorithm for CUBE MULTIPLICATION with running time $O(n^{4-\epsilon})$ would imply an algorithm for MAX-3SAT with running time $O^*(2^{n(1-\frac{\epsilon}{4})})$.

We discuss only the Williams' algorithm for MAX-2SAT and its generalization for MAX-3SAT in this chapter. It is possible to use the same idea to make generalizations for MAX-kSAT for any k, which involves a definition of k-dimensional arrays and their multiplication, but we won't discuss these

generalizations in this thesis.

3.1 The fast MAX-2SAT algorithm

Recall the MAX-2SAT problem:

Problem 5 (MAX-2SAT)**.**
Input: A 2-CNF formula ϕ having n variables and m clauses, and integer number $\tilde{k}$.
Task: Determine whether there exists a truth assignment to the variables of ϕ that satisfies at least $\tilde{k}$ clauses.

The MAX-2SAT problem is NP-complete. A naive algorithm for the MAX-2SAT problem would try all truth assignments to the variables of ϕ and for each assignment count the number of satisfied clauses. Thus such naive algorithm has time complexity $O^*(2^n)$. Now we will describe the fast exact exponential time algorithm for MAX-2SAT that was proposed by Williams.

Algorithm 1 (Fast MAX-2SAT algorithm)**.** Assume without loss of generality that every clause in ϕ has exactly two literals. Otherwise we duplicate literals in clauses that contain just one literal. This does not change the solution, and could increase the size of the formula only by a constant factor.

Assume without loss of generality that the number of variables is divisible by 3. Otherwise, we could add one or two dummy variables without modifying ϕ, and the solution won't change.

We split the set of all the variables in 3 equal parts of size $\frac{n}{3}$, and denote these parts by P^i, $i = 0, \ldots, 2$. This division also splits all the clauses of ϕ into classes $C_{\{i,j\}}$, where a clause c containing variables u and v is in class $C_{\{i,j\}}$ if $u \in P^i$ and $v \in P^j$, $i, j \in \{0, 1, 2\}$.

There are 6 such classes, and we denote by $\mathcal{C}$ the family of all these classes:

$$\begin{aligned}\mathcal{C} = \{&C_{\{0,0\}}, C_{\{0,1\}},\\ &C_{\{1,1\}}, C_{\{1,2\}},\\ &C_{\{2,2\}}, C_{\{2,0\}}\}.\end{aligned} \tag{3.1}$$

$\mathcal{C}$ could be partitioned into three subfamilies:

$$\mathcal{C} = \mathcal{C}_{\{0,1\}} \cup \mathcal{C}_{\{1,2\}} \cup \mathcal{C}_{\{2,0\}}, \tag{3.2}$$

where

$$\begin{aligned} \mathcal{C}_{\{0,1\}} &= \{C_{\{0,0\}}, C_{\{0,1\}}\}, \\ \mathcal{C}_{\{1,2\}} &= \{C_{\{1,1\}}, C_{\{1,2\}}\}, \\ \mathcal{C}_{\{2,0\}} &= \{C_{\{2,2\}}, C_{\{2,0\}}\}. \end{aligned} \tag{3.3}$$

By $\psi(\alpha, C)$ we define the number of clauses in a class C that are satisfied by a truth assignment (or a partial truth assignment) α to the variables in ϕ.

We build an auxiliary weighted graph $G = (V, E)$ in the following way. For each truth assignment to the variables in P^i the graph G has a node. We denote by V^i the set of all nodes that correspond to truth assignments to the variables in P^i. Then the number of nodes in G is equal to $3 \cdot 2^{\frac{n}{3}}$.

For each of three V^i parts of $A(G, \Sigma)$ we enumerate all nodes of this part from 1 to $d = 2^{\frac{n}{3}}$, so that we can denote a node by v_l^i (the node number l in V^i, $1 \leq l \leq d$).

Edges of G are all possible 2-element sets $\{x, y\}$ where $x \in V^i$, and $y \in V^{i+1 \pmod 3}$. So G has $3 \cdot 2^{\frac{2n}{3}}$ edges.

We note that each node of G corresponds to a partial truth assignment to the variables in ϕ. And any set of nodes, with no two nodes from the same set V^i, — such as an edge, or three nodes of a triangle — corresponds to a truth assignment (or a partial truth assignment) to the variables in ϕ. We denote by α^B a truth assignment that corresponds to a set of nodes B in G.

For any edge $\{x, y\}$, $x \in V^i$, $y \in V^j$, we define its σ-weight:

$$\sigma(\{x, y\}) = \sum_{C \in \mathcal{C}_{\{i,j\}}} \psi(\alpha^{\{x,y\}}, C). \tag{3.4}$$

From the construction of G follows that there is a bijection between triangles in G and truth assignments to the variables of ϕ. A triangle in G has three nodes x_i such that $x_i \in V^i, i = 0 \ldots 2$, and the set $\{x_0, x_1, x_2\}$ corresponds to a truth assignment to the variables in ϕ. Any truth assignment to the variables in ϕ corresponds to a set of three nodes in G, and these nodes also form a triangle in G by construction.

Lemma 4. *The σ-weight of a triangle $x_0x_1x_2$ in G is equal to the number of clauses satisfied by $\alpha^{\{x_0,x_1,x_2\}}$.*

Proof. Consider a triangle $x_0x_1x_2$ whose σ-weight is equal to t. Without loss of generality we number its nodes in such a way that $x_0 \in V_0$, $x_1 \in V_1$, and $x_2 \in V_2$. Let's calculate the number of clauses in ϕ that are satisfied by the corresponding truth assignment $\alpha^{\{x_0,x_1,x_2\}}$.

Each clause c can be in exactly one of the 6 classes in $\mathcal{C}$. The total number of satisfied clauses in ϕ by $\alpha^{\{x_0,x_1,x_2\}}$ is equal to

$$\begin{aligned}
&\sum_{C\in\mathcal{C}} \psi(\alpha^{\{x_0,x_1,x_2\}}, C) = \\
&\sum_{i=0}^{2} \sum_{C\in\mathcal{C}_{\{i,i+1 \pmod 3\}}} \psi(\alpha^{\{x_0,x_1,x_2\}}, C) = \\
&\sum_{i=0}^{2} \sum_{C\in\mathcal{C}_{\{i,i+1 \pmod 3\}}} \psi(\alpha^{\{x_i,x_{i+1 \pmod 3}\}}, C) = \\
&\sum_{i=0}^{2} \sigma(\{x_i, x_{i+1 \pmod 3}\}),
\end{aligned} \tag{3.5}$$

which is the σ-weight of the triangle, which is equal to t. □

For all values of t, $\tilde{k} \leq t \leq m$ we can try all possible partitions $\Sigma = (\sigma_{01}, \sigma_{12}, \sigma_{20})$ of t into 3 terms (that is $t = \sigma_{01} + \sigma_{12} + \sigma_{20}$). For each such partition we build a subgraph $A(G, \Sigma)$ with the following property. An edge $\{v_i^p, v_j^q\}$ from G remains in $A(G, \Sigma)$ only if $\sigma(\{v_i^p, v_j^q\}) = \sigma_{pq}$, $p, q = 0, \ldots, 2$, $q = p + 1 \pmod 3$, $i, j = 1, \ldots, d$. For every value of t the number of such partitions is at most t^3.The subgraph $A(G, \Sigma)$ can be constructed in time $O^*(2^{\frac{2n}{3}})$ by going through all edges of G. From this construction follows that there exists a triangle in $A(G, \Sigma)$ if and only if there exists a triangle of σ-weight t in G. It follows that there exists a truth assignment to the variables of ϕ that satisfies exactly t clauses if and only if there exists a partition Σ of t such that there exists a triangle in $A(G, \Sigma)$.

Now we explain how to determine whether $A(G, \Sigma)$ contains a triangle by using matrix multiplication. We build three $d \times d$ matrices, A^{12}, A^{10}, A^{02}, such that an element A_{ij}^{pq} is equal to 1 if there is an edge $\{v_i^p, v_j^q\}$ in $A(G, \Sigma)$, and is equal to 0 otherwise, for $(p, q) \in \{(1, 2), (1, 0), (0, 2)\}$ and $i, j = 1, \ldots, d$.

Lemma 5. *The value of $A_{ij}^{12} \cdot (A^{10}A^{02})_{ij}$ is equal to the number of triangles in $A(G, W)$ that contain nodes v_i^1 and v_j^2, $1 \leq i \leq d$, $1 \leq j \leq d$.*

Proof. Consider a multiplication

$$A_{ij}^{12} A_{ik}^{10} A_{kj}^{02} \tag{3.6}$$

for some fixed i, j, k, $1 \leq k \leq d$, $1 \leq i \leq d$, $1 \leq j \leq d$. It is equal to 1 if all the multipliers are equal to 1, which means that all three edges $\{v_i^1, v_j^2\}$, $\{v_i^1, v_k^0\}$, and $\{v_k^0, v_j^2\}$ are in $A(G, W)$. So the multiplication (3.6) is equal

to 1 if $A(G,W)$ contains a triangle $v_k^0 v_i^1 v_j^2$, and 0 otherwise. By definition of matrix multiplication each element of the matrix $(A^{10}A^{02})$ is the sum of multiplications:

$$(A^{10}A^{02})_{ij} = \sum_{k=1}^{d} A_{ik}^{10} A_{kj}^{02}. \tag{3.7}$$

If we multiply (3.7) by A_{ij}^{12} we get

$$\begin{aligned} A_{ij}^{12} \cdot (A^{10}A^{02})_{ij} &= A_{ij}^{12} \sum_{k=1}^{d} A_{ik}^{10} A_{kj}^{02} = \\ &= \sum_{k=1}^{d} A_{ij}^{12} A_{ik}^{10} A_{kj}^{02}, \end{aligned} \tag{3.8}$$

which is the sum of the multiplications (3.6) for all k, $1 \leq k \leq d$. Different values of k correspond to different nodes in V_0, and thus the summation (3.8) counts all triangles in $A(G,W)$ that contain the nodes v_i^1, and v_j^2. □

The multiplication of two $d \times d$ matrices can be computed in $O(d^\omega)$ time. Therefore every $A(G,\Sigma)$ can be tested for having at least one triangle in time $O^*(2^{\frac{\omega n}{3}})$.

Finally, we have the following algorithm. We construct the auxiliary graph G for the formula ϕ. Then, for every t, $\tilde{k} \leq t \leq m$, we try all possible partitions Σ. For each such partition we construct the graph $A(G,\Sigma)$. Then $A(G,\Sigma)$ is tested for triangles using matrix multiplication as shown in lemma 5. If there exists a triangle in $A(G,\Sigma)$, then there exists a truth assignment which satisfies t clauses, following from lemma 4.

The total running time of the algorithm is $O^*(m \cdot m^3(2^{\frac{\omega n}{3}} + 2^{\frac{2n}{3}})) = O^*(2^{\frac{\omega n}{3}})$. If $\omega = 3$ then the running time of the algorithm is $O^*(2^n)$. If we use a faster algorithm for MATRIX MULTIPLICATION, with $\omega = 3 - \epsilon$, then the running time of this algorithm is $O^*(2^{n(1-\frac{\epsilon}{3})})$, where $\epsilon > 0$.

3.2 MAX-3SAT: a promising algorithm

Recall the MAX-3SAT problem:

Problem 6 (MAX-3SAT)**.**
Input: A 3-CNF formula ϕ having n variables and m clauses, and integer number $\tilde{k}$.
Task: Determine whether there exists a truth assignment to the variables of ϕ that satisfies at least $\tilde{k}$ clauses.

The MAX-3SAT problem is NP-complete, and a naive algorithm for it has time complexity $O^*(2^n)$, the same as for MAX-2SAT. There are no known faster algorithms to date. The approach that was described before for MAX-2SAT could possibly be applied to the MAX-3SAT problem in a similar way. In the rest of this section we will show how to do such generalization.

As we noted before, at this point such generalization gives an alternative $O^*(2^n)$ algorithm for MAX-3SAT and could lead to an $O^*(2^{n(1-\frac{\epsilon}{4})})$ algorithm, provided that there exists an $O(n^{4-\epsilon})$ algorithm for CUBE MULTIPLICATION.

The algorithm for MAX-3SAT will follow the ideas from algorithm 1 for MAX-2SAT. In that algorithm variables were divided in 3 sets, but now we will divide them into 4 sets. Every edge of the graph in algorithm 1 was representing clauses from classes in a subfamily with a partial truth assignment to the variables. Since clauses now contain 3 literals, instead of the auxiliary graph we build an auxiliary 3-uniform hypergraph, and then every edge is a set of 3 nodes. Instead of triangles we search for hypersquares, that correspond to truth assignments. And to find the number of hypersquares we use cube multiplication instead of matrix multiplication.

3.2.1 The algorithm description

Algorithm 2 (Split and list MAX-3SAT algorithm)**.** Assume without loss of generality that every clause in ϕ has exactly three literals. Otherwise we duplicate literals in clauses that contain one or two literals. This does not change the solution, and could increase the size of the formula only by a constant factor.

Assume without loss of generality that the number of variables is divisible by 4. Otherwise, we could add one, two or three dummy variables without modifying ϕ, and the solution won't change.

We split the set of all the variables in 4 equal parts of size $\frac{n}{4}$, and denote these parts by P^i, $i = 0, \ldots, 3$. This division also splits all the clauses of ϕ into classes $C_{\{i,j,k\}}$, where a clause c containing variables u, v, and w is in class $C_{\{i,j,k\}}$ if $u \in P^i$, $v \in P^j$ and $w \in P^k$, $i, j, k \in \{0, 1, 2, 3\}$. There are 20 such classes, and we denote by $\mathcal{C}$ the family of all these classes:

$$\begin{aligned}\mathcal{C} = \{&C_{\{0,0,0\}}, C_{\{0,0,1\}}, C_{\{0,1,1\}}, C_{\{0,0,2\}}, C_{\{0,1,2\}},\\ &C_{\{1,1,1\}}, C_{\{1,1,2\}}, C_{\{1,2,2\}}, C_{\{1,1,3\}}, C_{\{1,2,3\}},\\ &C_{\{2,2,2\}}, C_{\{2,2,3\}}, C_{\{2,3,3\}}, C_{\{2,2,0\}}, C_{\{2,3,0\}},\\ &C_{\{3,3,3\}}, C_{\{3,3,0\}}, C_{\{3,0,0\}}, C_{\{3,3,1\}}, C_{\{3,0,1\}}\}.\end{aligned} \tag{3.9}$$

$\mathcal{C}$ could be partitioned into four subfamilies:

$$\mathcal{C} = \mathcal{C}_{\{0,1,2\}} \cup \mathcal{C}_{\{1,2,3\}} \cup \mathcal{C}_{\{2,3,0\}} \cup \mathcal{C}_{\{3,0,1\}}, \tag{3.10}$$

where

$$\begin{aligned}
\mathcal{C}_{\{0,1,2\}} &= \{C_{\{0,0,0\}}, C_{\{0,0,1\}}, C_{\{0,1,1\}}, C_{\{0,0,2\}}, C_{\{0,1,2\}}\},\\
\mathcal{C}_{\{1,2,3\}} &= \{C_{\{1,1,1\}}, C_{\{1,1,2\}}, C_{\{1,2,2\}}, C_{\{1,1,3\}}, C_{\{1,2,3\}}\},\\
\mathcal{C}_{\{2,3,0\}} &= \{C_{\{2,2,2\}}, C_{\{2,2,3\}}, C_{\{2,3,3\}}, C_{\{2,2,0\}}, C_{\{2,3,0\}}\},\\
\mathcal{C}_{\{3,0,1\}} &= \{C_{\{3,3,3\}}, C_{\{3,3,0\}}, C_{\{3,0,0\}}, C_{\{3,3,1\}}, C_{\{3,0,1\}}\}.
\end{aligned} \tag{3.11}$$

By $\psi(\alpha, C)$ we define the number of clauses in a class C that are satisfied by a truth assignment (or a partial truth assignment) α to the variables in ϕ.

We build an auxiliary weighted 3-uniform hypergraph $H = (V, E)$ in the following way. For each truth assignment to the variables in P^i the hypergraph H has a node. We denote by V^i the set of all nodes that correspond to truth assignments to the variables in P^i. Then the number of nodes in H is equal to $4 \cdot 2^{\frac{n}{4}}$.

For each of four V^i parts of $A(H, \Sigma)$ we enumerate all nodes of this part from 1 to $d = 2^{\frac{n}{4}}$, so that we can denote a node by v_l^i (the node number l in V^i, $1 \le l \le d$).

Hyperedges of H are all possible triples of the form $\{x, y, z\}$, where $x \in V^i$, $y \in V^{i+1 \pmod 4}$, and $z \in V^{i+2 \pmod 4}$. So H has $4 \cdot 2^{\frac{3n}{4}}$ hyperedges.

We note that each node of H corresponds to a partial truth assignment to the variables in ϕ. And any set of nodes, with no two nodes from the same set V^i, — such as a hyperedge, or four nodes of a hypersquare — corresponds to a truth assignment (or a partial truth assignment) to the variables in ϕ. We denote by α^B a truth assignment that corresponds to a set of nodes B in H.

For any hyperedge $\{x, y, z\}$, $x \in V^i$, $y \in V^j$, $z \in V^k$, we define its σ-weight:

$$\sigma(\{x, y, z\}) = \sum_{C \in \mathcal{C}_{\{i,j,k\}}} \psi(\alpha^{\{x,y,z\}}, C). \tag{3.12}$$

From the construction of H follows that there is a bijection between hypersquares in H and truth assignments to the variables of ϕ. A hypersquare in H has four nodes x_i such that $x_i \in V^i, i = 0 \ldots 3$, and a set $\{x_0, x_1, x_2, x_3\}$ corresponds to a truth assignment to the variables in ϕ. Any truth assignment to the variables in ϕ corresponds to a set of four nodes in H, and these nodes also form a hypersquare in H by construction.

Lemma 6. *The σ-weight of a hypersquare $x_0x_1x_2x_3$ in H is equal to the number of clauses satisfied by $\alpha^{\{x_0,x_1,x_2,x_3\}}$.*

Proof. Consider a hypersquare $x_0x_1x_2x_3$ whose σ-weight is equal to t. Without loss of generality we number its nodes in such a way that $x_0 \in V_0$,

$x_1 \in V_1$, $x_2 \in V_2$, and $x_3 \in V_3$. Let's calculate the number of clauses in ϕ that are satisfied by the corresponding truth assignment $\alpha^{\{x_0,x_1,x_2,x_3\}}$.

Each clause c can be in exactly one of the 20 classes in $\mathcal{C}$. The total number of satisfied clauses in ϕ by $\alpha^{\{x_0,x_1,x_2,x_3\}}$ is equal to

$$\begin{aligned}
&\sum_{C\in\mathcal{C}} \psi(\alpha^{\{x_0,x_1,x_2,x_3\}}, C) = \\
&\sum_{i=0}^{3} \sum_{C\in\mathcal{C}_{\{i,i+1 \pmod 4,i+2 \pmod 4\}}} \psi(\alpha^{\{x_0,x_1,x_2,x_3\}}, C) = \\
&\sum_{i=0}^{3} \sum_{C\in\mathcal{C}_{\{i,i+1 \pmod 4,i+2 \pmod 4\}}} \psi(\alpha^{\{x_i,x_{i+1 \pmod 4},x_{i+2 \pmod 4}\}}, C) = \\
&\sum_{i=0}^{3} \sigma(\{x_i, x_{i+1 \pmod 4}, x_{i+2 \pmod 4}\}),
\end{aligned} \tag{3.13}$$

which is the σ-weight of the hypersquare, which is equal to t. □

For all values of t, $\tilde{k} \leq t \leq m$ we can try all possible partitions $\Sigma = (\sigma_{012}, \sigma_{123}, \sigma_{230}, \sigma_{301})$ of t into 4 terms (that is $t = \sigma_{012}+\sigma_{123}+\sigma_{230}+\sigma_{301}$). For each such partition we build a subhypergraph $A(H, \Sigma)$ with the following property. A hyperedge $\{v_i^p, v_j^q, v_k^r\}$ from H remains in $A(H, \Sigma)$ only if $\sigma(\{v_i^p, v_j^q, v_k^r\}) = \sigma_{pqr}$, $p, q, r = 0, \ldots, 3$, $r = q + 1 \pmod 4 = p + 2 \pmod 4$, $i, j, k = 1, \ldots, d$. For every value of t the number of such partitions is at most t^4.The subhypergraph $A(H, \Sigma)$ can be constructed in time $O^*(2^{\frac{3n}{4}})$ by going through all hyperedges of H. From this construction follows that there exists a hypersquare in $A(H, \Sigma)$ if and only if there exists a hypersquare of σ-weight t in H. It follows that there exists a truth assignment to the variables of ϕ that satisfies exactly t clauses if and only if there exists a partition Σ of t such that there exists a hypersquare in $A(H, \Sigma)$.

Now we explain how to determine whether $A(H, \Sigma)$ contains a hypersquare by using cube multiplication. We build 4 d-cubes, A^{123}, A^{023}, A^{103}, A^{120}, such that an element A^{pqr}_{ijk} is equal to 1 if there is a hyperedge $\{v_i^p, v_j^q, v_k^r\}$ in $A(H, \Sigma)$, and is equal to 0 otherwise, for $(p, q, r) \in \{(1, 2, 3), (0, 2, 3), (1, 0, 3), (1, 2, 0)\}$ and $i, j, k = 1, \ldots, d$.

Lemma 7. *The value of $A^{123}_{ijk} \cdot (A^{023}A^{103}A^{120})_{ijk}$ is equal to the number of hypersquares in $A(H, W)$ that contain nodes v_i^1, v_j^2 and v_k^3, $1 \leq i \leq d$, $1 \leq j \leq d$, $1 \leq k \leq d$.*

Proof. Consider a multiplication

$$A^{123}_{ijk} A^{023}_{ljk} A^{103}_{ilk} A^{120}_{ijl} \tag{3.14}$$

for some fixed l, i, j, k, $1 \leq l \leq d$, $1 \leq i \leq d$, $1 \leq j \leq d$, $1 \leq k \leq d$. It is equal to 1 if all the multipliers are equal to 1, which means that all four hyperedges $\{v_i^1, v_j^2, v_k^3\}$, $\{v_l^0, v_j^2, v_k^3\}$, $\{v_i^1, v_l^0, v_k^3\}$ and $\{v_i^1, v_j^2, v_l^0\}$ are in $A(H, W)$. So the multiplication (3.14) is equal to 1 if $A(H, W)$ contains a hypersquare $v_l^0 v_i^1 v_j^2 v_k^3$, and 0 otherwise. By definition of cube multiplication each element of the cube $(A^{023}A^{103}A^{120})$ is the sum of multiplications:

$$(A^{023}A^{103}A^{120})_{ijk} = \sum_{l=1}^{d} A_{ljk}^{023} A_{ilk}^{103} A_{ijl}^{120}. \tag{3.15}$$

If we multiply (3.15) by A_{ijk}^{123} we get

$$\begin{aligned} A_{ijk}^{123} \cdot (A^{023}A^{103}A^{120})_{ijk} &= A_{ijk}^{123} \sum_{l=1}^{d} A_{ljk}^{023} A_{ilk}^{103} A_{ijl}^{120} = \\ &= \sum_{l=1}^{d} A_{ijk}^{123} A_{ljk}^{023} A_{ilk}^{103} A_{ijl}^{120}, \end{aligned} \tag{3.16}$$

which is the sum of the multiplications (3.14) for all l, $1 \leq l \leq d$. Different values of l correspond to different nodes in V_0, and thus the summation (3.16) counts all hypersquares in $A(H, W)$ that contain the nodes v_i^1, v_j^2 and v_k^3. □

Assuming the multiplication of three d-cubes can be computed in $O(d^\xi)$ time, every $A(H, \Sigma)$ can be tested for having at least one hypersquare in time $O^*(2^{\frac{\xi n}{4}})$.

Finally, we have the following algorithm. We construct the auxiliary hypergraph H for the formula ϕ. Then, for every t, $\tilde{k} \leq t \leq m$, we try all possible partitions Σ. For each such partition we construct the subhypergraph $A(H, \Sigma)$. Then $A(H, \Sigma)$ is tested for hypersquares using cube multiplication as shown in lemma 7. If there exists a hypersquare in $A(H, \Sigma)$, then there exists a truth assignment which satisfies t clauses, following from lemma 6.

The total running time of the algorithm is $O^*(m \cdot m^4(2^{\frac{\xi n}{4}} + 2^{\frac{3n}{4}})) = O^*(2^{\frac{\xi n}{4}})$. If $\xi = 4$ then the running time of the algorithm is $O^*(2^n)$. If there exists a faster algorithm for CUBE MULTIPLICATION, with $\xi = 4 - \epsilon$, then the running time of this algorithm is $O^*(2^{n(1-\frac{\epsilon}{4})})$.

3.2.2 A change to GF(2)

In algorithm 1 we could directly apply any of better than naive algorithms for MATRIX MULTIPLICATION that are known already, to get an algorithm for

MAX-2SAT with improved running time. In algorithm 2 we could apply only a naive algorithm for CUBE MULTIPLICATION, because there is no known better algorithm for CUBE MULTIPLICATION. One of the main purposes of this thesis is to try to find a better algorithm for CUBE MULTIPLICATION.

For the reasons that will become apparent later it is easier to search for a faster algorithm for CUBE MULTIPLICATION over GF(2) than for CUBE MULTIPLICATION with integer elements. But then we need to ensure that if we find a better algorithm for CUBE MULTIPLICATION over GF(2) that would still help us to find a better algorithm for MAX-3SAT.

Suppose we do cube multiplication only over GF(2) in algorithm 2. The cubes A^{123}, A^{023}, A^{103}, A^{120}, that we have defined in the algorithm, have elements equal only to 0 or 1 initially, so we can consider them as cubes over GF(2). If we compute a cube multiplication of these cubes, lemma 7 would imply that we are able to count the number of hypersquares containing nodes v_i^1, v_j^2 and v_k^3, $1 \leq i \leq d$, $1 \leq j \leq d$, $1 \leq k \leq d$, modulo 2. Thus, if the number of such hypersquares is even, which means there are two truth assignments to the variables of ϕ that are different only for 1/4 of the variables and satisfy at least $\tilde{k}$ clauses, the algorithm 2 could end up yielding incorrect NO solution for a YES instance of MAX-3SAT, because it may happen that other truth assignments do not satisfy at least $\tilde{k}$ clauses.

To resolve this problem we will reduce MAX-3SAT to a problem that has a unique optimal solution. Notice that if there is a unique optimal solution for the problem instance, we can use cube multiplication over GF(2) to get a correct answer with the algorithm 2. To reduce MAX-3SAT to the problem that has a unique optimal solution we will apply lemma 3 (the Isolation lemma). The reduction comes at the cost of making the algorithm randomized. This randomized algorithm yields a correct solution with a probability at least p, $0 < p < 1$, where p can be chosen at the cost of polynomial factor increase in running time.

3.2.3 A randomized algorithm

Algorithm 3 (Randomized MAX-3SAT algorithm)**.** Assume without loss of generality that every clause in ϕ has exactly three literals. Otherwise we duplicate literals in clauses that contain one or two literals. This does not change the solution, and could increase the size of the formula only by a constant factor.

Assume without a loss of generality that the number of variables is divisible by 4. Otherwise, we could add one, two or three dummy variables without modifying ϕ, and the solution won't change.

We define integer ρ-weight for each variable in ϕ independently and uniformly at random from $\{1, \ldots, N\}$, where N is some number greater than n that we can choose (for example $N = 10n$). For any assignment or partial assignment to the variables of ϕ we define its ρ-weight as the sum of ρ-weights of variables set to *true*. Let f be the maximum number of clauses that could be satisfied in formula ϕ. Let $\mathcal{F}$ be a set of truth assignments that satisfy exactly f clauses. By the Isolation lemma (lemma 3), with probability at least $1 - \frac{n}{N}$, there is a unique solution that satisfies the maximum number of clauses with the minimal ρ-weight.

We build the auxiliary hypergraph H in the same way as in algorithm 2, but additionally we define for each edge $\{x, y, z\}$ its ρ-weight $\rho(\{x, y, z\})$ which is equal to the sum of ρ-weights of partial assignments α^x, α^y and α^z. Note that a ρ-weight of a hypersquare is equal to the ρ-weight of a corresponding truth assignment multiplied by 3, because weight of each node is contributed trice, by each of three edges that include that node.

The next step is just as in algorithm 2, but we try combinations of both ρ-weights and σ-weights. We try every t, $\tilde{k} \leq t \leq m$, every partition $\Sigma = (\sigma_{012}, \sigma_{123}, \sigma_{230}, \sigma_{301})$ of t, every $s = 3s'$, $0 \leq s' \leq nN$, every partition $P = (\rho_{012}, \rho_{123}, \rho_{230}, \rho_{301})$ of s into 4 terms (so that $s = \rho_{012}+\rho_{123}+\rho_{230}+\rho_{301}$). For each such partition combination (Σ, P) we build a subhypergraph $A(H, \Sigma, P)$ with the following property. A hyperedge $\{v_i^p, v_j^q, v_k^r\}$ from H remains in $A(H, \Sigma, P)$ only if both $\sigma(\{v_i^p, v_j^q, v_k^r\}) = \sigma_{pqr}$ and $\rho(\{v_i^p, v_j^q, v_k^r\}) = \rho_{pqr}$, $p, q, r = 0, \ldots, 3$, $r = q + 1 \pmod 4 = p + 2 \pmod 4$, $i, j, k = 1, \ldots, d$. For every value of t the number of its partitions is at most t^4, and for every value of s the number of its partitions is at most s^4. The subhypergraph $A(H, \Sigma, P)$ can be constructed in time $O^*(2^{\frac{3n}{4}})$ by going through all hyperedges of H. From this construction follows that there exists a hypersquare in $A(H, \Sigma, P)$ if and only if there exists a hypersquare of σ-weight t and ρ-weight s in H. It follows that there exists a truth assignment to the variables of ϕ that satisfies exactly t clauses and has ρ-weight s if and only if there exist a partition Σ of t and a partition P of s such that there exists a hypersquare in $A(H, \Sigma, P)$.

In the same way as in algorithm 2 we determine whether $A(H, \Sigma, P)$ contains an odd number of hypersquares with nodes v_i^1, v_j^2 and v_k^3, $1 \leq i \leq d$, $1 \leq j \leq d$, $1 \leq k \leq d$, by using cube multiplication over GF(2) of d-cubes, A^{123}, A^{023}, A^{103}, A^{120}, which are defined in the same way as in algorithm 2.

Finally, we have the following algorithm. We construct the auxiliary hypergraph H for the formula ϕ. Then, for every t, $\tilde{k} \leq t \leq m$, we try all possible partitions Σ, and for every s, $s = 3s'$, $0 \leq s' \leq nN$, we try all possible partitions P. For each such partition combination we construct

the subhypergraph $A(H, \Sigma, P)$. Then $A(H, \Sigma, P)$ is tested for odd number of hypersquares using GF(2) cube multiplication. If there exists an odd number of hypersquares, then there exists an assignment which satisfies t clauses, following from lemma 6.

With probability at least $1 - \frac{n}{N}$ there is exactly one optimal solution with the minimum ρ-weight, thus easily discovered by checking for odd number of hypersquares. Therefore with at least the same probability the algorithm yields a correct solution.

The total running time of the algorithm is $O^*(m \cdot m^4 \cdot nN \cdot (nN)^4(2^{\frac{\xi n}{4}} + 2^{\frac{3n}{4}})) = O^*(2^{\frac{\xi n}{4}})$. If $\xi = 4$ then the running time of the algorithm is $O^*(2^n)$. If there exists a faster algorithm even for CUBE MULTIPLICATION over GF(2), with $\xi = 4 - \epsilon$, then the running time of this algorithm is $O^*(2^{n(1-\frac{\epsilon}{4})})$.

Note that a choice of N affects both the probability of a correct solution and the running time of the algorithm. For example, if we choose $N = 10n$, we get an algorithm that yields a correct solution with a probability at least 0.9.

Chapter 4

A Cube Multiplication study

Recall the CUBE MULTIPLICATION problem:

Problem 7 (CUBE MULTIPLICATION)**.**
Input: Three n-cubes A, B and C.
Task: Compute the cube multiplication (ABC).

If we directly apply the definition of cube multiplication we get a naive algorithm for CUBE MULTIPLICATION with a running time $O(n^4)$. We are motivated to find a better algorithm because anything faster, like $O(n^{4-\epsilon})$, will improve the upper bound for MAX-3SAT.

To the best of our knowledge the CUBE MULTIPLICATION problem has not previously been studied, but we can use its similarity to the MATRIX MULTIPLICATION problem to find a faster algorithm.

The simplest faster than naive algorithm for MATRIX MULTIPLICATION is the Strassen's algorithm, discovered by Strassen [1969], was explained in section 2.5 on page 15. We try to define a variant of Strassen's algorithm for CUBE MULTIPLICATION, assuming that such algorithm exists. This also yields a puzzle of discovering whether such algorithm exists. But at first we explain the same approach to find fast recursive algorithm for MATRIX MULTIPLICATION, that is, to discover the Strassen's algorithm, assuming we don't know its coefficients. This makes explanations for the CUBE MULTIPLICATION algorithm easier to follow. In both cases we arrive to instances of the same problem, namely SET SPAN, which we define in section 4.1 on page 39.

4.1 Discovering Strassen algorithm over GF(2)

Suppose we want to compute the matrix multiplication (AB) of two $n \times n$ matrices A and B over GF(2) using the recursive algorithm. In each step we partition A and B into 2×2 block matrices and compute their matrix multiplication treating each block as an element:

$$\begin{aligned}\begin{bmatrix}(AB)^{1,1} & (AB)^{1,2}\\ (AB)^{2,1} & (AB)^{2,2}\end{bmatrix} &= \begin{bmatrix}A^{1,1} & A^{1,2}\\ A^{2,1} & A^{2,2}\end{bmatrix}\begin{bmatrix}B^{1,1} & B^{1,2}\\ B^{2,1} & B^{2,2}\end{bmatrix} = \\ &= \begin{bmatrix}A^{1,1}B^{1,1} + A^{1,2}B^{2,1} & A^{1,1}B^{1,2} + A^{1,2}B^{2,2}\\ A^{2,1}B^{1,1} + A^{2,2}B^{2,1} & A^{2,1}B^{1,2} + A^{2,2}B^{2,2}\end{bmatrix}\end{aligned} \tag{4.1}$$

We want to determine whether it is possible to make only 7 recursive calls to MATRIX MULTIPLICATION instead of 8.

We need to compute four blocks:

$$\begin{aligned}(AB)^{1,1} &= A^{1,1}B^{1,1} + A^{1,2}B^{2,1},\\ (AB)^{1,2} &= A^{1,1}B^{1,2} + A^{1,2}B^{2,2},\\ (AB)^{2,1} &= A^{2,1}B^{1,1} + A^{2,2}B^{2,1},\\ (AB)^{2,2} &= A^{2,1}B^{1,2} + A^{2,2}B^{2,2}.\end{aligned} \tag{4.2}$$

They are expressed as sums of terms, and all of the terms are of the form $A^{i,j}B^{k,l}$, $i, j, k, l \in \{1, 2\}$. Therefore we restrict ourselves to multiplications of the form

$$(a_{1,1}A^{1,1}+a_{1,2}A^{1,2}+a_{2,1}A^{2,1}+a_{2,2}A^{2,2})(b_{1,1}B^{1,1}+b_{1,2}B^{1,2}+b_{2,1}B^{2,1}+b_{2,2}B^{2,2}), \tag{4.3}$$

where both multipliers are linear combinations of blocks with GF(2) coefficients $a_{i,j}$ and $b_{k,l}$, $i, j, k, l = 1, 2$. Notice that working over GF(2) limits the possibilities we have for choosing multiplications, because there are only 2 possible choices for each coefficient, and thus $(2^2)^2 = 256$ choices for such a multiplication.

Suppose we perform 7 such multiplications:

$$\begin{aligned}M^x &= \Big(\sum_{i=1,j=1}^{2} a^x_{i,j}A^{i,j}\Big)\Big(\sum_{k=1,l=1}^{2} b^x_{k,l}B^{k,l}\Big) = \\ &= \sum_{i=1,j=1}^{2}\sum_{k=1,l=1}^{2} a^x_{i,j}b^x_{k,l}A^{i,j}B^{k,l}, \quad x = 1, \ldots, 7.\end{aligned} \tag{4.4}$$

We are now interested whether it is possible to write each of the 4 blocks (4.2) as linear combinations of these 7 multiplications M^x that we have:

$$(AB)^{i,j} = \sum_{x=1}^{7} q_{i,j}^x M^x, \quad i,j = 1,2, \tag{4.5}$$

where $q_{i,j}^x$ are GF(2) coefficients.

Both M^x and $(AB)^{i,j}$, $x = 1,\ldots,7$, $i,j = 1,2$, are linear combinations of terms $A^{i,j}B^{k,l}$, $i,j,k,l \in \{1,2\}$, where there are 16 possible such terms. We can uniquely identify such a linear combination by a 16-dimensional GF(2) vector in the following way.

Consider the lexicographic ordering $\widetilde{(AB)}$ of these 16 terms:

$$\begin{aligned}(&A^{1,1}B^{1,1}, A^{1,1}B^{1,2}, A^{1,1}B^{2,1}, A^{1,1}B^{2,2}, A^{1,2}B^{1,1}, A^{1,2}B^{1,2}, A^{1,2}B^{2,1}, A^{1,2}B^{2,2},\\ &A^{2,1}B^{1,1}, A^{2,1}B^{1,2}, A^{2,1}B^{2,1}, A^{2,1}B^{2,2}, A^{2,2}B^{1,1}, A^{2,2}B^{1,2}, A^{2,2}B^{2,1}, A^{2,2}B^{2,2})\end{aligned}$$

In that ordering a term $A^{i,j}B^{k,l}$ has index $8(i-1)+4(j-1)+2(k-1)+l$. We call a 16-dimensional GF(2) vector $\hbar^L$ a *characteristic vector* of a linear combination L if

$$L = \sum_{i=1}^{16} \hbar_i^L \cdot \widetilde{(AB)}_i. \tag{4.6}$$

That is, each element of $\hbar^L$ is a coefficient for the corresponding term of the ordering $\widetilde{(AB)}$ in the linear combination L.

Then, each of the blocks (4.2) could be identified by its characteristic vector:

$$\begin{aligned}\hbar^{(AB)^{1,1}} &= [1000001000000000],\\ \hbar^{(AB)^{1,2}} &= [0100000100000000],\\ \hbar^{(AB)^{2,1}} &= [0000000010000010],\\ \hbar^{(AB)^{2,2}} &= [0000000001000001].\end{aligned} \tag{4.7}$$

And similarly each multiplication M^x could be identified by its characteristic vector $\hbar^{M^x}$, where the element of the vector $\hbar^{M^x}$ corresponding to a term $A^{i,j}B^{k,l}$ is equal to $a_{i,j}b_{k,l}$:

$$\begin{aligned}\hbar^{M^x}_{8(i-1)+4(j-1)+2(k-1)+l} &= a_{i,j}^x b_{k,l}^x,\\ x = 1,\ldots,7, &\quad i,j,k,l = 1,2.\end{aligned} \tag{4.8}$$

Note that for any two linear combinations L_1 and L_2 of terms from $\widetilde{(AB)}$ the following holds:

$$\hbar^{L_1+L_2} = \hbar^{L_1} + \hbar^{L_2}. \tag{4.9}$$

Therefore we can rewrite equation (4.5) in characteristic vector form:

$$\hbar^{(AB)^{i,j}} = \sum_{x=1}^{7} q_{i,j}^{x} \hbar^{M^x}, \quad i,j = 1,2, \tag{4.10}$$

From the equations (4.7), (4.8) and (4.10) follows a SAT-formulation for the puzzle of finding a Strassen's algorithm for MATRIX MULTIPLICATION. Every coefficient and vector element in these equations will have a corresponding boolean variable. We need to rewrite sum and product in terms of boolean operators and connect all the equations in one formula by boolean operators. Such construction is explained in details for the puzzle that corresponds to a faster CUBE MULTIPLICATION algorithm in section 4.2.3 on page 46. The construction for the puzzle we consider here is simpler and uses the same approach.

Note that equations (4.10) have a solution if and only if

$$(AB)^{i,j} \in span(\{M^1, \ldots, M^7\}), \quad i,j = 1,2. \tag{4.11}$$

We can calculate characteristic vectors of all 256 possible multiplications of the form (4.3), and then what we need is to choose 7 of them, such that (4.11) holds. Thus our puzzle could be viewed as an instance of the following problem which we call SET SPAN.

Problem 8 (SET SPAN)**.**
Input: Set $\bar{M}$ and set R of k-dimensional GF(2) vectors and number f.
Task: Determine whether there exists a subset M of size at most f, $M \subset \bar{M}$, such that

$$r \in span(M), \quad \forall r \in R, \tag{4.12}$$

that is, determine whether there exists a subset M that spans a vector space which contains all vectors from R.

Thus, to find the Strassen's algorithm, we need to solve an instance of SET SPAN with $R = \{\hbar^{(AB)^{1,1}}, \hbar^{(AB)^{1,2}}, \hbar^{(AB)^{2,1}}, \hbar^{(AB)^{2,2}}\}$ and $\bar{M}$ equal to the set of characteristic vectors of all 256 possible multiplications of the form (4.3), and $f = 7$.

We denote the puzzle of finding a variant of Straseen algorithm for MATRIX MULTIPLICATION that was discussed above by MATRIX-STRASSEN.

4.2 A generalization for cube multiplication

Let's define now a version of the Strassen's algorithm, generalized for CUBE MULTIPLICATION. For that we need to prove at first that block cube multiplication works in the same way it works for block matrix multiplication. Then we define a Strassen-like algorithm for CUBE MULTIPLICATION, assuming it exists, which in turn defines a puzzle of determining whether such algorithm exists. We do it in a similar way as in section 4.1. We do it for $2 \times 2 \times 2$ block partitioning, however it is possible to define such algorithm for any $m \times m \times m$ partitioning, but a size of the corresponding puzzle increases with increase of m. For the same reason we consider GF(2) cubes and coefficients. If the coefficients are integers, the number of possible combinations becomes infinite and it becomes even more difficult to determine whether one can choose proper coefficients.

4.2.1 A recursive algorithm

Lemma 8 (Block cube multiplication)**.** *Let A, B, C and D are n-cubes such that $D = (ABC)$, and $n = mo$. We partition A, B and C cubes into $m \times m \times m$ block cubes:*

$$A = \left[\begin{bmatrix} A^{1,1,1} & \dots & A^{1,m,1} \\ \vdots & \ddots & \vdots \\ A^{m,1,1} & \dots & A^{m,m,1} \end{bmatrix}, \cdots, \begin{bmatrix} A^{1,1,m} & \dots & A^{1,m,m} \\ \vdots & \ddots & \vdots \\ A^{m,1,m} & \dots & A^{m,m,m} \end{bmatrix}\right] \tag{4.13}$$

$$B = \left[\begin{bmatrix} B^{1,1,1} & \dots & B^{1,m,1} \\ \vdots & \ddots & \vdots \\ B^{m,1,1} & \dots & B^{m,m,1} \end{bmatrix}, \cdots, \begin{bmatrix} B^{1,1,m} & \dots & B^{1,m,m} \\ \vdots & \ddots & \vdots \\ B^{m,1,m} & \dots & B^{m,m,m} \end{bmatrix}\right] \tag{4.14}$$

$$C = \left[\begin{bmatrix} C^{1,1,1} & \dots & C^{1,m,1} \\ \vdots & \ddots & \vdots \\ C^{m,1,1} & \dots & C^{m,m,1} \end{bmatrix}, \cdots, \begin{bmatrix} C^{1,1,m} & \dots & C^{1,m,m} \\ \vdots & \ddots & \vdots \\ C^{m,1,m} & \dots & C^{m,m,m} \end{bmatrix}\right] \tag{4.15}$$

If we think of them as of m-cubes, we can calculate block cube E,

$$E = \left[\begin{bmatrix} E^{1,1,1} & \dots & E^{1,m,1} \\ \vdots & \ddots & \vdots \\ E^{m,1,1} & \dots & E^{m,m,1} \end{bmatrix}, \cdots, \begin{bmatrix} E^{1,1,m} & \dots & E^{1,m,m} \\ \vdots & \ddots & \vdots \\ E^{m,1,m} & \dots & E^{m,m,m} \end{bmatrix}\right] \tag{4.16}$$

as cube multiplication of A, B and C, where we treat each block as element in cube multiplication , that is

$$E^{p,q,r} = (ABC)^{p,q,r} = \sum_{h=1}^{m} A^{h,q,r} B^{p,h,r} C^{p,q,h}, \quad p, q, r = 1 \dots m. \tag{4.17}$$

Then elements of E will be the same as the corresponding elements of D, that is the block-wise cube multiplication yields the same result as the direct cube multiplication.

Proof. Let's show that for any choice of i, j and k an element $D_{i,j,k}$ is equal to the corresponding element of E, $i, j, k = 1, \ldots, n$. For each choice of i, j and k, $i, j, k = 1, \ldots, n$, there exist values p, q, r, s, t, u such that

$$\begin{aligned} i &= (p-1)o + s, \quad 1 \le p \le m, \quad 1 \le s \le o, \\ j &= (q-1)o + t, \quad 1 \le q \le m, \quad 1 \le t \le o, \\ k &= (r-1)o + u, \quad 1 \le r \le m, \quad 1 \le u \le o. \end{aligned} \tag{4.18}$$

Then an element $M_{i,j,k}$ of a cube M (where M can be any of the considered n-cubes) could be also referred by $M^{p,q,r}_{s,t,u}$ in the corresponding $m \times m \times m$ block cube. We need to show that $E^{p,q,r}_{s,t,u} = D_{i,j,k}$.

$$\begin{aligned} E^{p,q,r}_{s,t,u} &= (ABC)^{p,q,r}_{s,t,u} \overset{(a)}{=} \\ &\overset{(a)}{=} (\sum_{h=1}^{m} A^{h,q,r} B^{p,h,r} C^{p,q,h})_{s,t,u} = \\ &= \sum_{h=1}^{m} (A^{h,q,r} B^{p,h,r} C^{p,q,h})_{s,t,u} \overset{(b)}{=} \\ &\overset{(b)}{=} \sum_{h=1}^{m} \sum_{g=1}^{o} A^{h,q,r}_{g,t,u} B^{p,h,r}_{s,g,u} C^{p,q,h}_{s,t,g} \overset{(c)}{=} \\ &\overset{(c)}{=} \sum_{h=1}^{m} \sum_{g=1}^{o} A_{(h-1)o+g,j,k} B_{i,(h-1)o+g,k} C_{i,j,(h-1)o+g} \overset{(d)}{=} \\ &\overset{(d)}{=} \sum_{l=1}^{n} A_{l,j,k} B_{i,l,k} C_{i,j,l} = (ABC)_{i,j,k} \\ &= D_{i,j,k} \end{aligned} \tag{4.19}$$

In transition (a) we rewrite (ABC) using cube multiplication definition, in this case for blocks. In transition (b) we rewrite each cube multiplication of blocks using cube multiplication definition. In transition (c) we change indexing of elements of cubes A, B and C from block cube notation to cube notation. In transition (d) we change the indices h and g to l, $l = (h-1)o+g$, in the same way as in formulas (4.18). □

This gives us a recursive algorithm for CUBE MULTIPLICATION where in order to compute the multiplication of three $n \times n$ cubes we compute m^3 cube multiplications of three $\frac{n}{m}$-cubes. Assume without loss of generality

that $n = m^d$ for some d. Otherwise, we could add at most $(m-1)n$ rows, $(m-1)n$ columns and $(m-1)n$ planes filled with zeros to make n a power of m. If m is a constant that will increase the input size only by a constant factor.

Algorithm 4 (Recursive cube multiplication)**.** If $n > 1$, then partition cubes A, B and C into equally-sized block cubes, each cube into $m \times m \times m$ blocks. Calculate (ABC) block-wise recursively. If $n = 1$, return $A^{1,1,1}B^{1,1,1}C^{1,1,1}$.

Its running time is upper bounded by the following recurrence relation:

$$T(1) = O(1), \tag{4.20}$$
$$T(n) = m^4 T(n/m) + mO(n^3), \tag{4.21}$$

where m^4 recursive calls to CUBE MULTIPLICATION of blocks are accounted in the first term of the right side in the formula (4.21), and summation of them element-wise in the second term of the formula. Considering m as a constant, application of the master theorem shows that this recurrence give us an $O(n^{\log_m^{(m^4)}}) = O(n^4)$ algorithm, whose complexity is the same as of a naive algorithm. If we let $m = 2$, we need to make 16 recursive cube multiplications in that algorithm each time. Can we make 15 recursive cube multiplications, just like in Strassen's algorithm we do 7 recursive matrix multiplications instead of 8? In that case, the complexity of such algorithm would be $O(n^{\log_2^{(15)}}) = O(n^{3.91})$.

4.2.2 Discovering algorithm over GF(2)

If $m = 2$, we need to make 16 recursive calls in algorithm 4. We want to determine whether it is possible to make only 15 recursive calls.

We need to compute eight blocks:

$$\begin{aligned} &(ABC)^{i,j,k} = A^{1,j,k}B^{i,1,k}C^{i,j,1} + A^{2,j,k}B^{i,2,k}C^{i,j,2}, \\ &i,j,k = 0,1. \end{aligned} \tag{4.22}$$

They are expressed as sums of terms, and all of the terms are of the form $A^{i,j,k}B^{p,q,r}C^{s,t,u}$, $i,j,k,p,q,r,s,t,u \in \{1,2\}$. Therefore we restrict ourselves to multiplications of the form

$$(\sum_{i,j,k=1}^{2} a_{i,j,k}A^{i,j,k})(\sum_{p,q,r=1}^{2} b_{p,q,r}B^{p,q,r})(\sum_{s,t,u=1}^{2} c_{s,t,u}C^{s,t,u}), \tag{4.23}$$

where all three factors are linear combinations of blocks with GF(2) coefficients. Notice that working over GF(2) limits the possibilities we have for

choosing multiplications, because there are only 2 possible choices for each coefficient, and thus $2^{24} = 16777216$ choices for such a multiplication.

Suppose we perform 15 such multiplications:

$$\begin{aligned} M_x &= (\sum_{i,j,k=1}^{2} a^x_{i,j,k} A^{i,j,k})(\sum_{i,j,k=1}^{2} b^x_{i,j,k} B^{i,j,k})(\sum_{i,j,k=1}^{2} c^x_{i,j,k} C^{i,j,k}) = \\ &= \sum_{i,j,k=1}^{2} \sum_{p,q,r=1}^{2} \sum_{s,t,u=1}^{2} a^x_{i,j,k} b^x_{p,q,r} c^x_{s,t,u} A^{i,j,k} B^{p,q,r} C^{s,t,u}, \\ x &= 1, \ldots, 15. \end{aligned} \tag{4.24}$$

We are now interested whether it is possible to write each of the 8 blocks (4.22) as linear combinations of these 15 multiplications M^x that we have:

$$(ABC)^{i,j,k} = \sum_{x=1}^{15} q^x_{i,j,k} M^x, \quad i,j,k = 1,2, \tag{4.25}$$

where $q^x_{i,j,k}$ are GF(2) coefficients.

Both M^x and $(ABC)^{i,j,k}$, $x = 1, \ldots, 15$, $i,j,k = 1,2$, are linear combinations of terms $A^{i,j,k}B^{p,q,r}C^{s,t,u}$, $i,j,k,p,q,r,s,t,u \in \{1,2\}$, where there are $(2^3)^3 = 512$ possible such terms. We can uniquely identify such a linear combination by a 512-dimensional GF(2) vector in the following way.

Consider the lexicographic ordering $\widetilde{(ABC)}$ of these 512 terms:

$$\begin{aligned} &(A^{1,1,1}B^{1,1,1}C^{1,1,1}, \\ &A^{1,1,1}B^{1,1,1}C^{1,1,2}, \\ &A^{1,1,1}B^{1,1,1}C^{1,2,1}, \\ &A^{1,1,1}B^{1,1,1}C^{1,2,2}, \\ &A^{1,1,1}B^{1,1,1}C^{2,1,1}, \\ &\quad \ldots \\ &A^{2,2,2}B^{2,2,2}C^{1,2,2}, \\ &A^{2,2,2}B^{2,2,2}C^{2,1,1}, \\ &A^{2,2,2}B^{2,2,2}C^{2,1,2}, \\ &A^{2,2,2}B^{2,2,2}C^{2,2,1}, \\ &A^{2,2,2}B^{2,2,2}C^{2,2,2}) \end{aligned}$$

In that ordering a term $A^{i,j,k}B^{p,q,r}C^{s,t,u}$ has index

$$\begin{aligned} &256(i-1) + 128(j-1) + 64(k-1) + 32(p-1) + \\ &+ 16(q-1) + 8(r-1) + 4(s-1) + 2(t-1) + u. \end{aligned} \tag{4.26}$$

We call a 512-dimensional GF(2) vector $\hbar^L$ a *characteristic vector* of a linear combination L if

$$L = \sum_{i=1}^{512} \hbar_i^L \cdot \widetilde{(ABC)}_i. \tag{4.27}$$

That is, each element of $\hbar^L$ is a coefficient for the corresponding term of the ordering $\widetilde{(ABC)}$ in the linear combination L.

To make notation more concise we introduce the following alias for indices that are related to cube blocks:

$$\begin{aligned} &(i,j,k) \sim h, \\ &h = (i-1)*4 + (j-1)*2 + k, \\ &i,j,k = 1,2. \end{aligned} \tag{4.28}$$

We also introduce the following aliases for the indices (4.26) of the ordering $\widetilde{(ABC)}$:

$$\begin{aligned} &(i,j,k,p,q,r,s,t,u) \sim g, \\ &g = 256(i-1) + 128(j-1) + 64(k-1) + 32(p-1) + \\ &+ 16(q-1) + 8(r-1) + 4(s-1) + 2(t-1) + u, \\ &i,j,k,p,q,r,s,t,u = 1,2. \end{aligned} \tag{4.29}$$

and

$$\begin{aligned} &(i,j,k) \sim l, \\ &l = 64(i-1) + 8(j-1) + k, \\ &i,j,k = 1,8. \end{aligned} \tag{4.30}$$

Thus, an element $\widetilde{(ABC)}_{i,j,k,p,q,r,s,t,u}$ is a term $A^{i,j,k}B^{p,q,r}C^{s,t,u}$.

Then, each of the blocks (4.22) could be identified by its characteristic vector $\hbar^{(ABC)^{v,w,y}}$:

$$\begin{aligned} &\hbar^{(ABC)^{v,w,y}}_{i,j,k,p,q,r,s,t,u} = \begin{cases} 1, & \text{if } v=p=s, w=j=t, y=k=r, i=q=u, \\ 0, & \text{otherwise,} \end{cases} \\ &v,w,y,i,j,k,p,q,r,s,t,u = 0,1, \end{aligned} \tag{4.31}$$

that has exactly two elements which are equal to 1, and all other elements are equal to 0. Equation (4.31) follows directly from equation (4.22) and the alias definition (4.29), and guarantee that only two terms $(A^{1,w,y}B^{v,1,y}C^{v,w,1}$

and $A^{2,w,y}B^{v,2,y}C^{v,w,2}$, from definition of cube multiplication) indexed like $A^{i,j,k}B^{p,q,r}C^{s,t,u}$ have nonzero coefficient for each block $(ABC)^{v,w,y}$.

And similarly each multiplication M^x could be identified by its characteristic vector $\hbar^{M^x}$:

$$\begin{aligned} &\hbar^{M^x}_{i,j,k,p,q,r,s,t,u} = a^x_{i,j,k}b^x_{p,q,r}c^x_{s,t,u}, \\ &x = 1,\ldots,15, \quad i,j,k,p,q,r,s,t,u = 1,2. \end{aligned} \tag{4.32}$$

Note that for any two linear combinations L_1 and L_2 of terms from $\widetilde{(ABC)}$ the following holds:

$$\hbar^{L_1+L_2} = \hbar^{L_1} + \hbar^{L_2}. \tag{4.33}$$

Therefore we can rewrite equation (4.25) in characteristic vector form:

$$\hbar^{(ABC)^{i,j,k}} = \sum_{x=1}^{15} q^x_{i,j,k}\hbar^{M^x}, \quad i,j,k = 1,2, \tag{4.34}$$

From the equations (4.31), (4.32) and (4.34) follows a SAT-formulation for the puzzle of finding a variant of Strassen's algorithm for CUBE MULTIPLICATION. This construction is explained in details in section 4.2.3 on the following page.

Note that equations (4.34) have a solution if and only if

$$(AB)^{i,j,k} \in span(\{M^1,\ldots,M^{15}\}), \quad i,j,k = 1,2. \tag{4.35}$$

Our puzzle could be viewed as an instance of the SET SPAN problem (problem 8), as follows: $R = \bigcup_{i,j,k=1}^{2} \hbar^{(ABC)^{i,j,k}}$ and $\bar{M}$ is equal to the set of characteristic vectors of all 16777216 possible multiplications of the form (4.23), and $f = 15$.

We denote the puzzle of finding a variant of the Straseen's algorithm for CUBE MULTIPLICATION that was discussed above by CUBE-STRASSEN.

Assuming there exists a solution to equations (4.25) and (4.24) and we know it, we get the following Strassen-like algorithm for CUBE MULTIPLICATION over GF(2).

Algorithm 5 (Fast cube product)**.** If $n = 1$, return $A^{1,1,1}B^{1,1,1}C^{1,1,1}$. If $n > 1$, then partition cubes A, B and C into equally-sized $2 \times 2 \times 2$ block cubes

$$A = \left[\begin{bmatrix} A^{1,1,1} & A^{1,2,1} \\ A^{2,1,1} & A^{2,2,1} \end{bmatrix}, \begin{bmatrix} A^{1,1,2} & A^{1,2,2} \\ A^{2,1,2} & A^{2,2,2} \end{bmatrix}\right] \tag{4.36}$$

$$B = \left[\begin{bmatrix} B^{1,1,1} & B^{1,2,1} \\ B^{2,1,1} & B^{2,2,1} \end{bmatrix}, \begin{bmatrix} B^{1,1,2} & B^{1,2,2} \\ B^{2,1,2} & B^{2,2,2} \end{bmatrix}\right] \tag{4.37}$$

$$C = \left[\begin{bmatrix} C^{1,1,1} & C^{1,2,1} \\ C^{2,1,1} & C^{2,2,1} \end{bmatrix}, \begin{bmatrix} C^{1,1,2} & C^{1,2,2} \\ C^{2,1,2} & C^{2,2,2} \end{bmatrix}\right] \tag{4.38}$$

Calculate $D = (ABC)$ in the following way. We compute 15 n-cubes $M_x, \quad x = 1, \ldots, 15$:

$$M_x = (\sum_{i,j,k=1}^{2} a^x_{i,j,k} A^{i,j,k})(\sum_{i,j,k=1}^{2} b^x_{i,j,k} B^{i,j,k})(\sum_{i,j,k=1}^{2} c^x_{i,j,k} C^{i,j,k}), \tag{4.39}$$

where $a^x_{i,j,k}$, $b^x_{i,j,k}$, and $c^x_{i,j,k}$ are GF(2) coefficients that we have assumed to know. The cube multiplications of blocks are computed recursively. And then we compute each block of the resulting cube as a linear combination of M_x:

$$D^{i,j,k} = \sum_{x=1}^{15} q^x_{i,j,k} M_x, \quad i,j,k = 1, \ldots, 2, \tag{4.40}$$

where $q^x_{i,j,k}$ are GF(2) some coefficients that that we have assumed to know.

4.2.3 Satisfiability formulations

From the equations (4.31), (4.32) and (4.34) we could construct a boolean formula that is satisfiable if and only if there exists a solution that satisfy these equations. Such formula can be used as input for SAT solvers, which are computer programs that solve the satisfiability problem. Most of the solvers accept only CNF formulas as input, some may accept more general formulas. We explain here how to construct such a formula, both general and its CNF variant.

To do that, we rewrite every vector equation (4.34) into 512 equations written in terms of vector elements:

$$\hbar_l^{(ABC)^{i,j,k}} = \sum_{x=1}^{15} q^x_{i,j,k} \hbar_l^{M^x}, \tag{4.41}$$

$$l = 1, \ldots, 512, \quad i,j,k = 1,2. \tag{4.42}$$

Each vector element and each coefficient are GF(2) elements and could be treated as boolean variables. We then replace every multiplication with conjunction, and every addition with exclusive disjunction in all the equations:

$$\hbar_l^{(ABC)^{i,j,k}} = \bigoplus_{x=1}^{15} q^x_{i,j,k} \wedge \hbar_l^{M^x}, \tag{4.43}$$

$$l = 1, \ldots, 512, \quad i,j,k = 1,2. \tag{4.44}$$

The equations (4.32) can be rewritten then as

$$\begin{aligned} &\hbar^{M^x}_{i,j,k,p,q,r,s,t,u} = a^x_{i,j,k} \wedge b^x_{p,q,r} \wedge c^x_{s,t,u}, \\ &x = 1, \ldots, 15, \quad i, j, k, p, q, r, s, t, u = 1, 2. \end{aligned} \tag{4.45}$$

By substituting in formulas (4.34) elements of characteristic vectors with the corresponding right-hand boolean formulas in equations (4.32), and then by connecting all these formulas into one boolean formula by conjunction, considering (4.31), we get the following boolean formula:

$$\begin{aligned} \varphi &= \bigwedge_{i,j,k,l=1}^{8} \psi_{i,j,k,l}, \\ \psi_{i,j,k,l} &= \begin{cases} \bigoplus_{x=1}^{15} q^x_l \wedge a^x_i \wedge b^x_j \wedge c^x_k, & \text{if } \hbar^{(ABC)^l}_{i,j,k} = 1, \\ \neg(\bigoplus_{x=1}^{15} q^x_l \wedge a^x_i \wedge b^x_j \wedge c^x_k), & \text{otherwise} \end{cases} \\ &i, j, k, l = 1, \ldots, 8. \end{aligned} \tag{4.46}$$

In case we have restrictions for SAT-solver input, we may need to construct a formula in CNF form, or in CNF form with possible exclusive disjunction clauses.

To do this, we can use properties of boolean operators to transform the formula (4.46) into a CNF with the same number of variables. However, such formula could be very huge and inconvenient to construct.

Instead of substituting elements of characteristic vectors by formulas we may leave them as boolean variables, and change the equations to CNF formulas, possibly with introducing additional variables (see a paper by Tseitin [1983]). Let's consider an example. Each of formulas (4.45) is of the form $d = a \wedge b \wedge c$. We may change it by the following CNF formula:

$$(d \vee \neg a \vee \neg b \vee \neg c) \wedge (\neg d \vee a) \wedge (\neg d \vee b) \wedge (\neg d \vee c). \tag{4.47}$$

Any truth assignment to the variables a, b, c, and d evaluates both formulas to the same value.

In a similar way we can transform any of the formulas (4.43). They are of the form $c = (q_1 \wedge m_1) \oplus (q_2 \wedge m_2) \oplus \ldots \oplus (q_{15} \wedge m_{15})$. We note that $a = b_1 \oplus b_2 \oplus b_3 \oplus b_4 = b_1 \oplus (b_2 \oplus (b_3 \oplus b_4))$. We can introduce additional variables for the parts in parentheses. Every formula of the form $c = a \oplus b$ could be replaced by a formula

$$(\neg a \vee \neg b \vee \neg c) \wedge (\neg a \vee b \vee c) \wedge (a \vee \neg b \vee c) \wedge (a \vee b \vee \neg c). \tag{4.48}$$

And every formula of the form $c = a \wedge b$ could be replaced by a formula

$$(c \vee \neg a \vee \neg b) \wedge (\neg c \vee a) \wedge (\neg c \vee b). \tag{4.49}$$

Using these two transformations we can replace formulas (4.43) by CNF formulas. Note that we can introduce variables for bigger formulas, like $d = (q_1 \wedge m_1) \oplus (q_2 \wedge m_2)$, thus introducing fewer variables, but their corresponding CNF formulas may be bigger, having possibly more clauses and clauses of bigger size. There is a tradeoff between number of variables, number of clauses, and size of clauses.

For variables that correspond to values of $\hbar^{(ABC)^{i,j,k}}$ we get a clause with one literal that is positive if the value of a variable is 1, and negative otherwise.

By connecting with conjunction all CNF clauses we get from the transformations we get a CNF formula that can be used as input for almost any SAT solver.

In the way boolean formula was constructed from the equations (4.31), (4.32) and (4.34) for CUBE-STRASSEN, it could be constructed from the equations (4.7), (4.8) and (4.10) for MATRIX-STRASSEN. It involves fewer transformations, because it is simpler and smaller than the formula for CUBE-STRASSEN.

4.2.4 An algorithm for the set span problem

We have showed that we can determine whether a Strassen-like algorithm for CUBE MULTIPLICATION exists by solving an instance of the SET SPAN problem.

We build our instances of SET SPAN from trying to improve MATRIX MULTIPLICATION and CUBE MULTIPLICATION when doing recursive partitioning in blocks of size $\frac{n}{2}$. One could try improving them by using partitioning of size $\frac{n}{m}$. However the size of the puzzle we are trying to solve becomes significantly bigger with increase of m.

For the SET SPAN problem we denote by n the number of vectors in $\bar{M}$, and by s the number of vectors in R. We note that to witness that $r \in span(M)$ it is sufficient to find $|M|$ coefficients q_r^x such that

$$\sum_{x=1}^{|M|} q_r^x m^x = r. \tag{4.50}$$

This is a a system of linear equations. Thus, a fast way to check if a k-dimensional GF(2) vector is in the span of l vectors is to use the Gaussian elimination (see a book by Lay [2012]). The complexity of the Gaussian elimination is $O(l^2 k)$.

In a naive algorithm that solves the SET SPAN problem we can try all possible subsets M of size less or equal than f of the set $\bar{M}$ and check if the

vectors from R are in the span of M by trying all possible coefficients in equation (4.50). Number of ways to choose g vectors from a set of n vectors is $\binom{n}{g} = \frac{n!}{g!(n-g!)}$. To find if vectors from R are in span of M we can try all possible combinations of values of q_r^x coefficients and test equation (4.50). For every vector $r \in R$ it takes $O^*(2^g gk)$ time. The total running time of the naive algorithm is $O^*(\sum_{i=s}^{f}(\binom{n}{i} \cdot s2^i ik)) = O^*(\binom{n}{f}2^f sfk)$. Having guessed M, the coefficients q_r^x could be computed using Gaussian elimination in time $O(f^2k)$, thus improving the naive algorithm runtime to $O^*(\binom{n}{f}f^2ks)$.

The number of possible subsets checked in the naive algorithm is quite huge for CUBE-STRASSEN, it is close to $2.35 \cdot 10^{108}$.

Now we prove that there exists a faster algorithm that solves the SET SPAN problem. Instead of trying all subsets of size f it tries sets of smaller size (thus decreasing number of possible subsets checked) and does slightly more work for each such set. Overall it has a better running time.

Assume that there exists a solution set M. Without loss of generality we assume that M is a set of linearly independent vectors. If not, we can remove linearly dependent vectors from it and the resulting set will still be a solution of a smaller size. Because we choose a solution from subsets of size up to f, linear independent solutions of smaller size will be rediscovered anyway. Without loss of generality we assume that R is a set of linearly independent vectors (note that this is the case for the MATRIX-STRASSEN and CUBE-STRASSEN instances). If R is not a set of linearly independent vectors, we could substitute it by a basis of R. That does not change the solution. By lemma 2 on page 20 there exists a set P, $P \subseteq M$, such that $P \cup R$ is a basis of $span(M)$. Instead of trying to find a set M, we can try to find a set P. A simple way would be to try all possible subsets P. For every such P we know the span of $P \cup R$. We need to find a basis for that span, consisting of vectors from $\bar{M}$. We could find a set $\hat{M}$ of all vectors from $\bar{M}$ that are in $span(P \cup R)$. We know that $M \subseteq \hat{M}$, and thus $span(M) = span(P \cup R) = span(\hat{M})$. Therefore any basis of $span(\hat{M})$ that consists of vectors from $\hat{M}$ is also a basis for $P \cup R$, and thus is a solution of SET SPAN.

We can improve the last step by searching for a basis of $\hat{M}$ directly, without building $\hat{M}$. Each time we discover a vector from $\hat{M}$, we add it to a set $\check{M}$ (initially empty) if this does not make $\check{M}$ linearly dependent.

So we have the following algorithm.

Algorithm 6 (Check-Span-Fast)**.** For all subsets P of size t, $0 \leq s \leq (f-s)$ we do the following procedure. If $P \cup R$ is linearly dependent, we continue to the next set. Otherwise, we create an empty set of solution vectors $\check{M}$, and for all vectors in $\bar{M}$ do the following. If a vector v, $v \in \bar{M}$, is in the span of

$P \cup R$, then it may be one of solution vectors. We add it to $\check{M}$ if it does not make $\check{M}$ linearly dependent, that is if v is not in the span of $\check{M}$. Span check can be done using Gaussian elimination. If at some point the size of $\check{M}$ is equal to f, then we have found a solution. Otherwise, there is no solution that contains P. Indeed, if we have a set $\check{M}$ of less than f vectors from $\bar{M}$ and any other vector from $\bar{M}$ that lies in the span of $P \cup R$ can be expressed as a linear combination of vectors from $\check{M}$, then there is no basis of $P \cup R$ consisting of vectors from $\bar{M}$. If we assume such basis does exist, then we can express each of it's vectors as a linear combination of vectors in $\check{M}$, which is a contradiction to our assumption that it's a basis. This algorithm has worst case complexity $O(\sum_{i=0}^{f-s} \binom{n}{i} \cdot n(f^2k + f^2k)) = O(\binom{n}{f-s} nf^2k)$.

Thus, we can conclude from discussion above the following theorem.

Theorem 1. *There exists an algorithm for* SET SPAN *with running time* $O(\binom{n}{f-s} nf^2k)$.

In fact, it is possible to improve the running time a bit more, getting an $O(\binom{n}{f-s} nfk)$ algorithm. We explain how that was done in section 5.2.3 on page 58.

Instances of the SET SPAN problem that correspond to fast CUBE MULTIPLICATION or MATRIX MULTIPLICATION algorithms have numbers f, r and k much smaller than n. Indeed, $r = m^d$, $f < m \cdot m^d$, $k = m^{d^2}$, $n = 2^{dm^d}$ in case of CUBE MULTIPLICATION or MATRIX MULTIPLICATION, where d is a number of array dimensions (2 for matrix, 3 for cube), m is a size of block partitioning. In case of CUBE MULTIPLICATION these numbers are:

$$r = 8, \tag{4.51}$$
$$f = 15, \tag{4.52}$$
$$k = 512, \tag{4.53}$$
$$n = 16777216. \tag{4.54}$$

The Check-Span-Fast algorithm therefore is substantially faster compared to the naive algorithm. Still, the number of checked subsets is close to $3.75 \cdot 10^{50}$, which is a very big number, so it is not realistic to try all possible subsets, and we should rely on future improvements or heuristics.

Chapter 5

Computational experiments

Now, when we have formulated the problem of finding fast block multiplication algorithms for MATRIX MULTIPLICATION and CUBE MULTIPLICATION, the next step is to try to use computers to solve them. For this, we need to prepare input for SAT solvers, or to make a computer program that can solve the instances of SET SPAN that correspond to finding fast MATRIX MULTIPLICATION and CUBE MULTIPLICATION algorithms.

5.1 Using SAT solvers to solve the puzzles

5.1.1 Conducted experiments

The SAT formulations that we discussed in section 4.2.3 could be used as input in SAT-solvers which are computer programs that solve SAT instances. There are many of SAT solvers, and most common input format for SAT solvers is *dimacs* text format that encodes a CNF formula. Some solvers can handle exclusive disjunction clauses in the input formula.

Encoding of a problem into an instance of SAT and then into a CNF formula is itself a problem, because running time of a SAT solver depends on this encoding and for different formulas and sat solvers an efficient encoding could be different for every particular problem and not straightforward, corresponding to the article by Björk [2009] about sat encoding. In our thesis we have made three simple and straightforward encodings for CUBE-STRASSEN and one for MATRIX-STRASSEN, using techniques we have described in section 4.2.3 on page 46, in hope that one of them may lead to

a reasonably fast discovery of a solution by a SAT solver.

The encoding for MATRIX-STRASSEN has 1028 variables, 3280 clauses, and 8592 literals.

For CUBE-STRASSEN we have made one encoding that contains exclusive disjunction clauses and two CNF encodings. In one of the CNF encodings, φ_1, we have introduced many additional variables, so that every clause has most three literals. φ_2 has 127904 variables, 443712 clauses, and 1182784 literals. In another CNF encoding we have introduced fewer variables at the cost of having bigger clauses and bigger formula size. It has 65504 variables, 391168 clauses, and 1281024 literals. A formula with exclusive disjunction clauses, φ_3, has 66016 variables, 311296 clauses, and 860160 literals. However, we haven't managed to get a solution for CUBE-STRASSEN in acceptable time with any of the solvers we have used, so we cannot say which of the encodings is better.

To encode the problems and write it in *dimacs* text format we have developed a computer program in Python programming language. It is quite straightforward and builds formulas step by step, by creating variables and making clauses that reflect equations, as described in section 4.2.3 on page 46. We have developed an additional program in Python to test our boolean formula transformations for correctness.

We have used several open source sat-solvers:

- lingeling and its parallel versions plingeling and treengeling (see a paper by Biere [2014]), which won in several categories in SAT Competition 2014 and 2013.

- cryptominisat, a parallel sat-solver that can handle exclusive disjunction clauses (see a paper by Soos [2014]), which are the case for CUBE-STRASSEN.

All of these SAT solvers have managed to solve the MATRIX-STRASSEN instance in a matter of seconds on a modern computer.

We have tried to run lingeling, plingeling, treengeling, and cryptominisat on encodings φ_1 and φ_2, and cryptominisat on encoding φ_3 on a modern computer for at least 24 hours of CPU time, but without getting any result, positive or negative. We also have tried to run plingeling, treengeling, and cryptominisat on φ_1 and φ_2 encodings, and cryptominisat on φ_3 encoding on a parallel supercomputer with 80 cores and 130 gb of RAM, each for over 24 CPU hours, but also with no result.

5.1.2 Ideas for possible improvements

We have got several ideas that could help to find a solution using SAT solvers, but we didn't have time to implement them.

One direction is to make a more efficient encoding for CUBE-STRASSEN. A good overview of encoding techniques could be found in the article by Björk [2009] A possible improvements we see are:

- Add additional clauses that encode redundant constraints. This could speed up a solver, as well as slow it down. For example, we could encode a constraint that $\prod_{x=1}^{15} q_{i,j,k}^{x} \neq 0$, since any result block is non zero.
- In our formulation we didn't take account of ordering, that is several solutions could be permutations of variable indices. One could add additional constraints that restrict solution to be ordered, so we only search for one solution from a set of equivalent solutions.
- One could research whether it is possible to take into account symmetry of cubes.

In general, any additional properties of the instance could be taken into account and possibly encoded as additional constraints. From the other side, an encoding can be seen from perspective of a solver. Thus, by knowing better how a particular solver works, one could modify encoding or add constraints that will help the solver to find a solution in more efficient way. Or one may even change a way the problem is encoded in general.

Another direction is to learn more about SAT solvers and tune parameters of a solver according to the algorithms that the solver uses and a used encoding.

5.2 A solver and heuristics for set span instances

5.2.1 The basic algorithm

We have developed a program that implements the Check-Span-Fast algorithm (algorithm 6) with some optimizations for two instances of SET SPAN, MATRIX-STRASSEN and CUBE-STRASSEN, using C++ programming language. We will denote this program by Set-Span-Checker. Our choice of programming language is connected to the possibility to use templates in C++ and its fast speed.

The Check-Span-Fast algorithm consists of two parts: selection part and checking part. In the selection part we select a set P of at most $(f - s)$ vectors, and in the checking part we check if P leads to a correct solution. By solution space we further mean a set of all such sets P, and by a candidate solution we mean such a set P.

From now on we assume that we consider only sets P of size exactly $(f - s)$, that is we search for a solution set $\check{M}$ of size 15 for CUBE-STRASSEN or size 7 for MATRIX-STRASSEN. Indeed if there is a solution with smaller number of vectors, we can add a linearly independent vector from $\bar{M}$ to it and it will still be a solution.

The implementation of Check-Span-Fast was done in the following way. In the beginning we calculate all vectors in the set $\bar{M}$, that is 2^{24} vectors for CUBE-STRASSEN (or 2^8 vectors for MATRIX-STRASSEN), that correspond to an instance we are working with, and a set of result vectors R. In the selection part we select all possible sets of $(f - s)$ vectors, and for each such set we run the checking part. The checking part in general is implemented as explained in algorithm 6 on page 49. Several optimizations improve running time of this part, and they are explained in section 5.2.3 on page 57.

We have tested this algorithm on MATRIX-STRASSEN, and on a modern computer it manages to find a solution (and thus a variant of the Strassen's algorithm), in under one minute, by going through all the solution space.

5.2.2 Heuristics and parallelization

In case of CUBE-STRASSEN, the solution space is very big and it is not possible to check it fully in a reasonable time. Thus, we have developed several heuristics for selecting candidate solutions. Unlike in the exact algorithm Check-Span-Fast, we only try some of all possible candidates, in hope to find a solution.

We also have parallelized our program to use the advantage of multi-core computers. We have done it in a trivial way by using OpenMP application programming interface. Several processes run the same heuristic and share a memory, so that the set $\bar{M}$ is computed only once and is used by all the processes. When we say further that we run tests on a parallel computer we mean that we use the parallel implementation for that.

At first, we have started with the simplest possible heuristic that selects a random candidate each time. We have tested it on MATRIX-STRASSEN, and it finds a solution after 2500 candidate checks on average (from 1000 runs of the heuristic), spending less than a second of CPU time.

However, it didn't work that well with CUBE-STRASSEN. Even after searching for several days and trying several hundred thousands of candidates

it didn't find a solution. An average time to check one candidate was approximately one second by one process.

We have decided to give each candidate a score, which is equal to the number of vectors in $\check{M}$ in the end of checking part of Check-Span-Fast, that is the rank of $\hat{M}$, a set of vectors from $\bar{M}$ which are in the span of $P \cup R$. So the lowest possible score for CUBE-STRASSEN is 7, and the highest possible score is 15. In case of MATRIX-STRASSEN the lowest possible score is 3 and the highest possible score is 7. So we want to find a candidate that has a highest possible score.

We have defined a neighbourhood relation for candidates as follows. Two candidates P_1 and P_2 are *neighbours* if and only if:

1. $P_1 = P_2 \setminus \hbar_1 \cup \hbar_2$, that is candidate P_1 has vector $\hbar_1$ and candidate P_2 has vector $\hbar_2$, and the rest is the same for both;

2. corresponding block coefficients for $\hbar_1$ and $\hbar_2$ are the same except for one.

The second statement means (in case of CUBE-STRASSEN) that if $\hbar_1$ corresponds to a linear combination

$$(\sum_{i,j,k=1}^{2} a_{i,j,k}A^{i,j,k})(\sum_{p,q,r=1}^{2} b_{p,q,r}B^{p,q,r})(\sum_{s,t,u=1}^{2} c_{s,t,u}C^{s,t,u}),$$

then changing any of the 24 coefficients $a_{i,j,k}$, $b_{p,q,r}$ or $c_{s,t,u}$ in this linear combination will make a linear combination with a such characteristic vector $\hbar_2$. A *neighbourhood* of a candidate is a set of candidates that are neighbours of that candidate. Therefore, each candidate can have at most $7 \cdot 24 = 168$ neighbours in this neighbourhood. By changing candidates to their neighbours we can move from any candidate to any other candidate in the solution space. For MATRIX-STRASSEN there are at most $3 \cdot 8 = 24$ neighbours for each candidate.

Using our scoring function and neighbourhood relation, we have implemented a *hill climbing metaheuristic*, that is a heuristic that selects a random candidate solution and then moves to the neighbour of that candidate which has the highest score among all neighbours of that candidate (or to random highest scoring neighbour from the set of such highest scoring neighbours, if there are several highest scoring neighbours), and then repeat it again and again, moving in solution space. By *iteration* we mean a change from a current candidate to its neighbour.

So we have the following hill climbing algorithm:

Algorithm 7 (Hill climbing)**.**

Step 1 Select a random candidate P and compute its score.

Step 2 Make a set $N[P]$ of all its neighbours.

Step 3 Compute scores for every candidate in $N[P]$.

Step 4 Assign to P one of the best scoring neighbours in $N[P]$ and go to **Step 2**.

If a solution is found during **Step 1** or **Step 3**, than stop and return it.

Because we can stuck in a local optimum (in terms of score) and/or on a plateau (when score is not improving), we do restart after each γ consecutive iterations without improving current score, by selecting a new random starting candidate.

In case of MATRIX-STRASSEN, the hill climbing heuristic with a relatively large γ yields a solution after doing approximately 0.005 restarts. These restarts are probably due to local optimum, because the number of restarts didn't change significantly for the values of γ equal to 10000, 100000, and 1000000. After trying different values of γ we have found that $\gamma = 9$ yields a solution by checking the smallest number of candidates on average. In that case on average 444 candidates are checked, 1 restart is done and 21 iterations are done. This is five times faster than the random search heuristic.

In case of CUBE-STRASSEN, the hill climbing heuristic doesn't work that well either. After running it for several days with a γ equal to 100, 10 and 2, no solution were found, and even the best found score was 7, the minimum possible score.

We have decided to combine the random search heuristic and the hill climbing heuristic in the following way. We ran the random search heuristic on a parallel computer with 80 cores for a week, and have collected all candidates that had a better score than 7. We have found 92 different candidates with a score equal to 8. Then, we have tried to use these 92 candidates as possible starting candidates in the hill climbing heuristic instead of selecting a random candidate. We ran the hill climbing heuristic with $\gamma = 5$ and number of restarts limited by 5 for each core on a 80-core computer (thus, 400 restarts in total, each of 5 iterations). Unfortunately, this approach also didn't help to find any candidate with a better score than 8, the solver was moving between candidates with score 8, finding hundreds of thousands of such candidates.

We have gathered some properties of the solution space for MATRIX-STRASSEN. From 1873200 candidates in the solution space 7998 candidates lead to linearly dependent sets $P \cup R$. For the other candidates score is distributed as follows:

1. 689640 candidates have score 3
2. 800190 candidates have score 4
3. 336492 candidates have score 5
4. 38160 candidates have score 6
5. 720 candidates have score 7

5.2.3 Optimizations

The most time-consuming task in Set-Span-Checker is a checking part, which we denote by Check-Vectors. It is executed repeatedly for different candidate sets. So our objective is to optimize Check-Vectors as much as possible.

The Check-Vectors subroutine uses a subroutine to check if a vector is in the span of a set of vectors. In the basic algorithm we have used a Gaussian elimination algorithm to check if a vector is in the span of a set of vectors. To speed up the calculations we have used several optimizations, that are discussed further.

Also we check if a set $P \cup R$ is linearly independent in the beginning of Check-Vectors. This is combined with a precomputing step that is described further.

Equation (4.50) that witnesses that vector r is in the span of M can be seen as a system of linear equations written in matrix form, $Ax = b$, where $A = [m^1, \ldots, m^{|M|}]$, $x = [q_r^1, \ldots, q_r^{|M|}]$, and $b = r$. In Gaussian elimination we use row operations on the augmented matrix $A|b$, that is adding one row to another, or swapping rows, to transform A into a row echelon form (while treating b as an additional column of A). In row echelon form the lower left part of A contains only zeros, and all of the zero rows are below

the non-zero rows:

$$\begin{bmatrix} A_{1,1} & A_{1,2} & \dots & A_{1,|M|-1} & A_{1,|M|} \\ 0 & A_{2,2} & \dots & A_{1,|M|-1} & A_{1,|M|} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & A_{|M|-1,|M|-1} & A_{|M|-1,|M|} \\ 0 & 0 & \dots & 0 & A_{|M|,|M|} \\ 0 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & 0 \end{bmatrix} \tag{5.1}$$

To check whether b is in span of M we need to check if elements of b that lie in all-zero rows of A (rows from $|M|+1$ to k) are equal to zero.

Preprocessing

Because Gaussian elimination is executed a lot of times for each candidate set, we have tried to make it as fast as possible. We need to check if a vector is in the span of a set of vectors in two places in Check-Vectors. The first place is when we check each vector v from $\bar{M}$ for being in the span of $P \cup R$. The second place is when we check a vector $v \in span(P \cup R)$ for being in the span of $\check{M}$. In both cases a lot of different vectors can be checked for being in the span of the same set of vectors. Thus, we can try to find which work we can do once for a fixed set of vectors, and thus a fixed matrix A (when we consider an equation $Ax = b$ for different b).

At first, we have decided to precompute the steps of Gaussian elimination, that is to remember each row operation properties, such as row indices, in a data structure. Since row operations do not depend on a vector b it is possible to do it once for a matrix A and then apply the recorded operations on each of different vectors b. That gave us a substantial speedup.

Then, we have tried another algorithm to determine whether a vector r is in the span of M. If we consider the corresponding matrix equation $Ax = b$, we note that column operations (that is adding one column to another, or swapping them) change the corresponding set of vectors, but such set will be a basis of the $span(M)$ (otherwise the original set would be linearly dependent, which is a contradiction). So we can use column operations to transform A into A' in a way that make $|M|$ rows in A' having all elements equal to 0 except for one, such that every column has a corresponding row with 1 at this column's index. That is

$$\begin{aligned} &\forall j = 1, \dots, |M|, \exists i_j, 1 \le i_j \le k : \\ &A'_{i_j,j} = 1, A'_{i_j,l} = 0, l = 1, \dots, |M|, l \neq j. \end{aligned} \tag{5.2}$$

Thus, for a vector b we can calculate a vector b':

$$b' = \sum_{j=1}^{|M|} A'_j \cdot b_{i_j}, \tag{5.3}$$

where i_j is taken from the equation (5.2), and A'_j is a column j of the matrix A'. If $b = b'$ than b is in the span of columns of A', and thus in the span of columns af A, which we wanted to check.

For that algorithm we also can remember each column operation properties — column's row indices i_j — in a data structure. Since column operations do not depend on a vector b it is possible to do it once for a matrix A and then use the recorded indices with each of different vectors b, using only $O(k|M|)$ operations. Also, when a new vector is added to M, it takes only $O(k|M|)$ operations to transform the matrix A' into A'' that satisfies (5.2), thus also updating the structure. Therefore, the complexity of this implementation of Check-Vectors is $O(f^2k+n(fk+fk+fk)) = O(nfk)$, because $f << n$. In the complexity formula the first term fk corresponds to checking if a vector v is in $P \cup R$, the second term fk corresponds to checking if v is in $\check{M}$, and the third term fk corresponds to updating of $\check{M}$ matrix structure. That means, we have improved the worst case complexity of the Check-Span-Fast algorithm a little, making it equal to $O(\binom{n}{f-s}nfk)$

The second approach of precomputing was faster in practice.

Bit checks

Before calculation of b' and comparing it to b we do one additional check to discard vectors that are not in span of M. In precomputing step we compute negation of logical or of all vectors from M:

$$M_{NOR} = \neg \bigcup_{v \in M} v. \tag{5.4}$$

Then, for an arbitrary vector v, if $M_{NOR} \wedge v$ has at least one bit equal to 1 then v does not lie in span of M. Indeed, such one at some index i means that all of vectors from M has zeros at this index and v has one at this index. Any vector in the span of M has therefore zero at index i and thus v is not in the span of M.

This check speeds up computations by nearly 50 percent.

Caching of Check-Vectors results

We have added a cache that stores results of Check-Vectors, so if does not compute everything again for the same candidate. It was implemented using

a `map` C++ structure, and gave a slight improvement in rare cases when the same candidate was checked again.

Optimizations that didn't work

We have tried also to check first for number of 1's in vectors b' and b before comparing them (since we can precompute that function for all vectors in $\bar{M}$ during initialization of the program), but that did not give any significant speed up.

We also have tried to use memoization by storing already calculated sums b' for different tuples $(b_{l_1}, \ldots, b_{l_{|M|}})$, but this also didn't give any significant speed up.

We have tried also to store a number of additional bits in all vectors, such that each bit is a linear combination of other bits of the same vector, and such combination is the same for all k-dimensional vectors we work with. We used these vectors just like the original ones, except that during comparison of b' and b we first checked these additional bits, and if they were the same, only then we checked original bits. If original vectors are not the same, than there is some probability that additional parts are not the same, but if original vectors are equal than additional part is always equal. However, this technique also didn't improve the computation speed.

5.2.4 Ideas for possible improvements

For the Check-Span-Fast algorithm, its implementation Set-Span-Checker and heuristics discussed above, one could try to improve them as follows:

- Try another score function. Maybe a function with a greater range of values, so there is less chance to stuck in a plateau.
- Try another neighbourhood type. For example, changing two coefficients instead of one. That will lead to a bigger neighbourhood for a candidate (at most $168 \cdot 168 = 28224$ neighbours, long time to check but more chance to find a better score). Or, change one coefficient at most for each of the factor cubes, so then we have at most $7 \cdot (9^3 - 1) = 3584$ neighbours.
- Try other metaheuristics, such as simulated annealing, tabu search, genetic algorithms, etc.
- Try to research whether it is possible to take into account symmetry of cubes.

- Heuristically limit solution space for CUBE-STRASSEN, by seeking similarities with more studied solution space of MATRIX-STRASSEN.

5.3 Notes on programs correctness

The source codes of the programs described above can be found on the following webpages:

`https://github.com/nasedil/checkspanfast`: the repository of the Set-Span-Checker and the heuristics.

`https://github.com/nasedil/strassen-sat-cube`: the repository of the SAT formulation generators.

Although we cannot be certain about the correctness of the Set-Span-Checker program and the heuristics, we believe that there is a low probability of serious mistakes or bugs in them. In case of MATRIX-STRASSEN we have done some testing to see if results were actually a variants of the Strassen's algorithm, like checking manually several results, and automated testing using several different algorithms for checking if a vector is in a span of a set of vectors, an comparing their results. For CUBE-STRASSEN we have tried to triple check that all generalizations in code are a proper generalisations of the MATRIX-STRASSEN case. Still, there could be some bugs related to memory management, since C++ does not have automatic memory management. A parallel code is quite trivial, so it is unlikely to be a trouble in that part of program. In SAT formula generator we have tested all our boolean formula transformations for correctness, and manually checked several results of SAT-solver output for MATRIX-STRASSEN that it was actually a variant of Strassen's algorithm. And again, we triple checked the CUBE-STRASSEN code for correctness.

Chapter 6

Conclusions

In this thesis we studied MAX-2SAT and MAX-3SAT problems and non-trivial algorithms for them, defining the auxiliary CUBE MULTIPLICATION problem. We have showed how an $O(n^{4-\epsilon})$ algorithm for CUBE MULTIPLICATION would imply a faster than naive $O^*(2^{n(1-\frac{\epsilon}{4})})$ algorithm for MAX-3SAT. For this we have defined a new problem, CUBE MULTIPLICATION.

In our attempt to find a faster than naive algorithm for CUBE MULTIPLICATION we have done the following:

- defined the SET SPAN problem;
- explained SAT solver formulations for two instance of the problem of finding a better recursive MATRIX MULTIPLICATION and CUBE MULTIPLICATION algorithms;
- implemented and explained the programs in Python for generating SAT solver formulations;
- explained the fast algorithm for SET SPAN
- implemented and explained the fast algorithm for SET SPAN in C++;
- implemented in C++ and explained several heuristics for MATRIX-STRASSEN and CUBE-STRASSEN;
- conducted and explained several computational experiments.

Although we didn't manage to find an $O(n^{4-\epsilon})$ Strassen-like algorithm for CUBE MULTIPLICATION, our algorithms and code may possibly be improved and reused to further attack the problem. We also showed how it is possible to discover variants of Strassen's algorithm using the proposed approaches and their implementation.

6.1 Future research

One could possibly generalize the approach we have used for MAX-3SAT algorithm to MAX-kSAT with larger values of k. It could involve a similar operation to MATRIX MULTIPLICATION and CUBE MULTIPLICATION, but on k k-dimensional arrays.

One can experiment with SAT encodings and SAT solvers, as proposed in section 5.1.2 on page 53.

One can further improve the solver we have developed for SET SPAN, as proposed in section 5.2.4 on page 60.

One can also try to make an Integer Linear Programming (ILP) formulation for CUBE-STRASSEN and try to solve it available ILP solvers.

Another approach would be to analyse cube multiplication operation from mathematical point of view. For that a good starting point would be to read articles about matrix multiplication and algorithms for matrix multiplication.

Bibliography

Armin Biere. Yet another local search solver and lingeling and friends entering the sat competition 2014. *SAT COMPETITION 2014*, pages 39–40, 2014.

Magnus Björk. Successful sat encoding techniques. *JSAT Addendum*, 2009.

Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

Sanjoy Dasgupta, Christos H Papadimitriou, and Umesh Vazirani. *Algorithms*. McGraw-Hill, Inc., 2006.

Fedor V Fomin and Dieter Kratsch. *Exact exponential algorithms*. Springer Science & Business Media, 2010.

François Le Gall. Powers of tensors and fast matrix multiplication. *arXiv preprint arXiv:1401.7714*, 2014.

Michael R Garey, David S. Johnson, and Larry Stockmeyer. Some simplified np-complete graph problems. *Theoretical computer science*, 1(3):237–267, 1976.

Nathan Jacobson. *Basic algebra I*. Courier Corporation, 2012.

Richard M Karp. *Reducibility among combinatorial problems*. Springer, 1972.

Jon Kleinberg and Éva Tardos. *Algorithm design*. Pearson Education India, 2006.

Melven R Krom. The decision problem for a class of first-order formulas in which all disjunctions are binary. *Mathematical Logic Quarterly*, 13(1-2): 15–20, 1967.

David C Lay. *Linear algebra and its applications*. Pearson, 2012.

Leonid A Levin. Universal sequential search problems. *Problemy Peredachi Informatsii*, 9(3):115–116, 1973.

Ketan Mulmuley, Umesh V Vazirani, and Vijay V Vazirani. Matching is as easy as matrix inversion. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 345–354. ACM, 1987.

Ramamohan Paturi, Pavel Pudlák, Michael E Saks, and Francis Zane. An improved exponential-time algorithm for k-sat. *Journal of the ACM (JACM)*, 52(3):337–364, 2005.

Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2012.

Mate Soos. Cryptominisat v4. *SAT COMPETITION 2014*, page 23, 2014.

Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.

Grigori S Tseitin. On the complexity of derivation in propositional calculus. In *Automation of reasoning*, pages 466–483. Springer, 1983.

J Eldon Whitesitt. *Boolean algebra and its applications*. Courier Corporation, 1995.

R Ryan Williams. *Algorithms and resource requirements for fundamental problems*. PhD thesis, Carnegie Mellon University Pittsburgh, PA, 2007.